The Joy of Pex
v1.2

Dan Gindikin, Peter Yianilos, Joe Kilian

December 12, 2011

Abstract

Pex is a preprocessor and a build management system for Pyrex, or Cython. Among other things, Pex adds the ability to conveniently write C fast numerics using `numpy.ndarray`, frees you from Makefiles and header files, and makes your Pyrex classes serializable, through both pickling and a faster scheme. To the user, Pex looks like a programming language that is much like Python, but with additional syntax, through which it can be made to run as fast as C in settings all the way from small numerical loops to large scale systems.

Pyrex is a Python to C compiler with the added functionality of C fast function calls, an object system with C fast attribute access and method invocation, and extra syntax that allows mixing-in of C code. Cython is a close cousin to Pyrex, which adds many convenient features. Most of the functionality of the language is in Pyrex, and its author Greg Ewing deserves the majority of the credit.

Contents

1 Introduction **5**
1.1 What You Are Assumed to Know 5
1.2 A Word of Caution 5
1.3 Heroic Example 6

2 Notation **7**

3 Getting Started **7**
3.1 Simple Example 7
3.2 Slightly Less Simple Example 8

4 Conventions **9**

5 Interacting with Python **9**

6 The `main()` function **10**

7 Pimports **11**
7.1 Simple 11
7.2 Elaborate 12

8 NDArray Decoration **13**
8.1 Simple 13
8.2 Elaborate 14

9 Modest Cdef Classes **26**
9.1 Simple 26
9.2 Elaborate 28

10 `pex.pexruntime` **45**

11 `pex_create_uninitialized()` **45**

12 Primitive C Types and Literals **46**

13 C `struct` and `typedef` **48**

14 Configuring Compilation **50**
14.1 Pragmas and the `scope` Object 52
14.2 pragma_c_only 52

15 Linking with External C Code **54**

16 Backtraces on `SIGSEGV`, `SIGFPE`, `SIGBUS`, and `SIGABRT` **55**

17 Command Line Usage **56**

18 Compiling for Code Coverage **57**

19 Distributing Code **58**

20 Exceptions **59**
20.1 Using Exceptions in Fast, Low Level Code 60
20.2 Exception Traceback Formatting 61

21 Efficiency **62**
21.1 Annotate Mode 62

22 Gotchas **63**

23 Crossplatform Status **69**

24 Acknowledgements **69**

25 Conclusion **69**

A Wrapping `cdef` functions **71**
A.1 Wrapping `cdef` functions: the slow and simple way 71
A.2 Wrapping `cdef` functions via classes 72
A.3 Obtaining generality via subclassing 72
A.4 Do we need all of these classes? 73
A.5 Wrapping Python objects as Pex objects 75

B What is Pyrex? **76**
B.1 Simple 76
B.2 Elaborate 76

C Differences Between Pex and Pyrex **82**
C.1 Defining cdef Class Attributes 82
C.2 Unsupported Features 83

D Pragmas **83**

E Builtins **85**

1 Introduction

Python is a wonderful language that manages to guide you to correct expression of complex algorithms through little code. Python has a great standard library, convenient fundamental datatypes, clean and pleasant syntax, and an import mechanism with just it time compilation (to bytecode), which proves one can have complex systems without makefiles or a complicated build process. But it's slow. On average, 50-100 times slower than C, for numerical code 100-400 times slower.

We felt that in principle there was no reason for Python to be this slow, and that by giving up very little of what makes Python great, one could make most of this speed penalty go away. Greg Ewing proved 95% of this point by writing Pyrex. However, in some ways Pyrex did not feel very *pythonic*, there was a lot of overhead to writing in Pyrex that was absent in Python. In order to write a complex system in Pyrex, one has to spend a lot of time on header files and Makefiles, much like with a C project. One likely also has to write a significant amount of custom code to enable objects to be serialized to disk, to print and to compare them. In our experience with complex projects, this boilerplate soaks up a large fraction of the effort, all the more painful since none of it has anything to do with the essence of the project. Pex generates all of this boilerplate for you, and then also augments Pyrex to allow you to write C fast numerics with a standard array datatype that is fully supported in Python – numpy's ndarray (see 8).

1.1 What You Are Assumed to Know

This document is not meant to be standalone: some knowledge of Python is assumed. A great deal of functionality of Pex attempts to match Python semantics. Pex is mostly Pyrex, so if you are not familiar with it, look through Appendix B and refer to Pyrex documentation as needed. It may also be useful to skim Cython documentation, though it does not have much at the moment (recall that Cython is a fork of the Pyrex project that Pex actually uses). We will also refer to numpy, which is a substantial numerical library for python.

1.2 A Word of Caution

You should be aware that Pex is not yet a full language. This early stage of Pex was always intended as a thin preprocessor layer for Pyrex. While immediately useful, this is our first, learning, attempt at capturing a blend of Python elegance and C performance. You'll no doubt encounter difficult to chase down compilation errors and strange limitations, many stemming from the fact that Pex does not employ a parser, but rather gets by on regular expressions. Throughout this document we will attempt to give you a flavor of these gotchas.

1.3 Heroic Example

The reason to put up with Pex's and Pyrex's various quirks is that you can get massive performance improvements for not much difference in code. The sweep algorithm implemented in Pex is about 75 times faster than the Python implementation for a 10x10 matrix, about 400 times faster for matrices of size 100x100 and above; as you can see in the diff of the two codes, the differences are minimal. This example, while heroic, is not contrived, for numerics code you can expect to see two orders of magnitude improvement in running time.

```
import numpy

def sweep(x):

    n = x.shape[0]



    l = 0
    u = n-1

    g_k=numpy.zeros(n,'double')

    pivot_product = 1.0

    for k in range(l,u+1):
        if x[k,k] == 0.0:
            pivot_product = 0.0
            break
        pivot_product *= x[k,k]
        x[k,k] = -1.0 / x[k,k]
        for j in range(n):
            if j == k:
                continue
            g_k[j] = x[j,k]
            x[j,k] = x[k,j] = -x[k,k] * g_k[j]
        for i in range(n):
            if i == k:
                continue
            for j in range(i+1):
                if j == k:
                    continue
                x[j,i] -= g_k[i] * x[k,j]
                x[i,j] = x[j,i]

    return pivot_product
```

```
import numpy

def sweep(ndarray<double 2d> x):

    cdef int    n,l,u,i,j,k
    cdef double pivot_product

    n = x.dimensions[0]
    l = 0
    u = n-1

    cdef ndarray<double,n> g_k

    pivot_product = 1.0

    for k from l <= k <= u:
        if x{k,k} == 0.0:
            pivot_product = 0.0
            break
        pivot_product *= x{k,k}
        x{k,k} = -1.0 / x{k,k}
        for j from 0 <= j < n:
            if j == k:
                continue
            g_k{j} = x{j,k}
            x{j,k} = x{k,j} = -x{k,k} * g_k{j}
        for i from 0 <= i < n:
            if i == k:
                continue
            for j from 0 <= j <= i:
                if j == k:
                    continue
                x{j,i} -= g_k{i} * x{k,j}
                x{i,j} = x{j,i}

    return pivot_product
```

2 Notation

Code snippets are given like so:

```
# lives in file "main.px"
print "hello world"
```

and unless otherwise specified are assumed to live in the file `main.px` (`.px` means a Pex file). Shell commands are prefixed with a "$", thus to run the above snippet, we say:

```
$ pex main.px
hello world
```

Usually we will just show the resulting output

```
out hello world
```

On occasion it is convenient to specify an entire directory tree, which we will do like so:

```
pkg/__init__.py

pkg/mod.py
        a_string = "foo bar baz"

main.px
        from pkg import mod

        print mod.a_string
```

Here we have a `main.px` file in the current directory, and a directory `pkg` with an empty file `__init__.py` and a Python module file `mod.py`. The resulting output when `main.px` is ran, is:

```
out foo bar baz
out
```

Unless otherwise specified, if there is a `main.px` or `main.py`, it's what is ran.

3 Getting Started

3.1 Simple Example

The venerable "hello world" program in Pex:

```
def main():
        print "hello world"
```

```
out
out hello world
```

We ran it with "pex main.px" (see 2). Here is what happens behind the scenes:

(i.) Pex creates files `main.pyx` and `main.pxd` from `main.px`

(ii.) Pyrex produces `main.c` from `main.pyx` and `main.pxd`

(iii.) from here the C toolchain takes over:

(a) gcc makes `main.o` from `main.c`

(b) gcc makes `main.so` from `main.o`. The file `main.so` is an importable Python module.

(iv.) now that `main.so` is made, Pex does the exact equivalent of the following Python code

```
#!/usr/bin/python
import main
main.main()
```

Essentially, after generating the Pyrex `.pyx`, `.pxd` files from the `.px` file, Pex ran the following sequence of commands:

```
$ pyrex -I main.pyx

$ gcc -c -fPIC -O6 -fno-strict-aliasing -I/usr/include/python2.3 main.c

$ gcc -shared -lm main.o -o main.so

$ python -c "import main; main.main()"
```

3.2 Slightly Less Simple Example

Here, in accordance with software engineering best practices, we split the functionality of `hello world` into two modules:

```
mod.px
        def func():
                return "hello world"

main.px
```

```
6         %pimport mod
7
8         def main():
9                 print mod.func()
```

```
out
out hello world
```

The `main` module, uses `%pimport` to bring in another module, called `mod` and call its function (see 7 about pimports). To see what happened behind the scenes, in excruciating detail, run `pex -vall -F main.px` (the `-vall` flag sets verbosity to all, `-F` forces a rebuild of everything).

4 Conventions

Throughout the manual, we will be referring to code sections running at "C speed" or "Python speed". Take `x=x+3` for example. If `x` is a C int, this is effectively a C expression – it runs at C speed, but if `x` is a python object, this code runs at Python speed, getting translated into Python library calls along the lines of `PyNumber_Add(x,PyInt_FromLong(3))`. As a rough estimate, Python speed is 1-2 orders of magnitude slower than C, usually closer to 2.

We will also refer to compile time and runtime. Compile time is everything that happens to turn a `.px` file into a `.so`. Runtime is when the `.so` is loaded by Python, and your code is actually executing.

5 Interacting with Python

In Pex, you can import Python modules, and use them just as you would in Python. In particular here is an example that shows you how to access the command line:

```
1 import sys
2
3 def main():
4
5         if sys.argv[1]=='Marco':
6                 print 'Polo'
7
8         elif sys.argv[1]=='Polo':
9                 print 'Marco'
```

running it from the shell we have:

```
$ pex main.px
Traceback (most recent call last):
```

```
  File "main.pyx", line 71, in main.main
    if sys.argv[1]=='Marco':   ## main.px,5

$ pex main.px Marco
Polo

$ pex main.px Polo
Marco
```

6 The main() function

The main() function has the same semantics as Python's if __name__=='__main__': clause. If your module has a main() function, and is executed from the command line, your main() is called. If your module is imported by something, main() is not called; for example if you have

```
1 mod.px
2         def main():
3                 print "hello world"
4
5 main.px
6         %pimport mod
```

and run pex main.px, mod's main() is never called.

Code in the topmost scope is executed at import time, just like in Python, so in the following example, we do get output:

```
1 mod.px
2         print "hello world"
3
4 main.px
5         %pimport mod
```

```
out
out hello world
```

:-(Do not write if __name__=='__main__': in a Pex file, Pyrex defines __name__, but it never defines it to be "__main__", thus this if statement will happily compile, but will never be true.

7 Pimports

7.1 Simple

Imports work in Pex in the same way they do in Python, except you say `%pimport` instead of `import`. In the simplest form you say,

```
%pimport <module_name>
```

You may also use more complicated forms, like in Python, the full syntax is the following:

```
%[from <package_or_module>] pimport <name> [as <alias>] [,<name> [as <alias>]]*
```

thus any of the following is valid:

```
pkg/__init__.py

pkg/submod.px

        def func():
                pass

        cdef class Klass:
                pass

mod.px

        cdef func():
                pass

        class Klass:
                pass

main.px

        %pimport mod, pkg.submod

        %pimport mod as foo, pkg.submod as bar

        %from pkg.submod pimport func, Klass as baz

        %from mod pimport func as gah, Klass
```

To import a Pex module from Python, do:

```
#!/usr/bin/python
import pex
mod=pex.pimport("mod")  # this imports the file mod.px
```

Imports are a weak point for Pex, Pyrex, and to some extent Python. There are some problem cases, some with errors easy to figure out, some not. Make sure you read the Gotchas section 7.2.2 below.

7.2 Elaborate

7.2.1 Search Path

In addition to honoring `PYTHONPATH` environment variable when searching for modules, pex also pays attention to the `PEXPATH` environment variable, which has the same semantics as `PYTHONPATH`.

7.2.2 Gotchas

:-(Like for Python, your Pex package directories need to contain an `__init__.py` file in order to be importable, an empty `__init__.py` file is sufficient. This is really easy to forget, and really annoying to discover, so be careful.

:-(This is a trick for troubleshooting your imports. Sometimes you can have name collisions between modules, and because Python memoizes imports by module name, your import will not give you the module object you expect. Suppose at some point earlier in the program, the module `foo` was imported, all later `import foo` statements will return this module, even if you are attempting to import a different `foo`, and according to the semantics of the search path can expect to get it. When this happens, you'll start seeing errors like "module object doesn't have expected attribute", etc. A useful way of debugging is to just `print foo` at the point of the error, which shows what file the module came from, and `print dir(foo)`, which shows the module's attributes.

A particularly nasty, and not uncommon, manifestation of this problem is when `foo` is a directory that happens to have an `__init__.py` file, which makes the directory importable. Worse yet, an `__init__.pyc` alone is also sufficient. To fix, remove `__init__.py` AND `__init__.pyc`.

:-(Can not do `pimport *`.

:-(If you wish to subclass a class that you import from another module, you must pimport that module without using `from` or `as`; do this:

```
1 %pimport pkg.mod
2 cdef class derived(pkg.mod.base_class):
```

```
3     pass
```

not this:

```
1 %from pkg pimport mod
2 cdef class derived(mod.base_class):
3     pass
```

:-(Can not do `pimport <module>.<var>`, can only do `pimport <module>` and then refer to the variable as `<module>.<var>`, or `from <module> pimport <var>`.

:-(Though `%pimport` looks like a regular statement, it is not, it causes things to happen both at compile time and at runtime. This means that the following piece of code will not do what you may expect:

```
1 if 0:
2     %pimport mod
```

The `if 0:` is purely runtime statement, it does not actually bind the `%pimport`. Here, the `%pimport` is still processed at compile time, but not at runtime, and bad things will happen.

8 NDArray Decoration

8.1 Simple

In Python, [] accesses are slow. For example, make yourself a numpy array (henceforth referred to as "ndarray"), and then access it like so:

```
1 arr = numpy.zeros(10)
2 arr[1]
```

There is a lot of work that happens behind the scenes for that `arr[1]`, here is just the partial C class stack:

```
PyObject_Call
PyEval_CallObjectWithKeywords
PyObject_GetItem
```

All this overhead makes the [] access two orders of magnitude slower than if `arr` was a C `double arr[]` array. Pex lets you get that speed back by saying `arr{1}` instead, though before you are allowed to do so, you must "decorate" `arr` with some type information:

```
cdef ndarray<double,10> arr
arr{1}
```

this `arr{1}` happens at C speed, as if `arr` was a C `int *`. Similarly for multiple dimensions:

```
def main():
        cdef ndarray<int,(2,3,4)> mat

        cdef int i,j,k

        for 0 <= i < dimlen(mat,0):
                for 0 <= j < dimlen(mat,1):
                        for 0 <= k < dimlen(mat,2):

                                mat{i,j,k} = i+j+k

        print mat
```

```
out
out [[[0 1 2 3]
out   [1 2 3 4]
out   [2 3 4 5]]
out
out  [[1 2 3 4]
out   [2 3 4 5]
out   [3 4 5 6]]]
```

Supported types for a decorated ndarray are `char,short,int,double`, and `object`.

8.2 Elaborate

8.2.1 Decorating NDArrays

There are a few ways ndarrays may become decorated. The simplest is when an ndarray comes in as an argument to a function:

```
def func(ndarray<short 2d> arr)
```

The general syntax is `ndarray<TYPE DIMENSIONALITYd>`. In the above example a 2 dimensional array of shorts is passed in. The type and dimensionality are checked, and if you try to pass in the wrong thing, an `NDArrayDecorationError` exception will be thrown.

There are also two ways to declare decorated ndarrays, the first is non-allocating:

```
cdef ndarray<double 1d> arr
```

The intention is that at some point later, the variable will be assigned an ndarray of the right type and dimensionality. Such assignments are typechecked, and an exception will be raised if the type or dimensionality is wrong. If you access your array before allocating it with something like `arr = numpy.zeros(3,'d')`, you will SEGFAULT (see 22 for complete discussion of this type of error).

The other flavor of decorated declaration is allocating:

DEATH!

```
cdef ndarray<int,3>        arr
cdef ndarray<char,(3,17)> arr2d
```

The above is equivalent to

```
cdef ndarray<int 1d>       arr   = numpy.zeros(3,'l')
cdef ndarray<char 2d>      arr2d = numpy.zeros((3,17),'b')
```

The syntax for one dimensional ndarray declaration is `ndarray<TYPE, DIMENSION1>`, for multi-dimensional: `ndarray<TYPE, (DIMENSION1,...,DIMENSIONn)>`.

The last way to decorate an ndarray is the following:

```
%decorate(arr, <double 3d>)
```

It is provided as a hack, to work around the fact that Pex does only shallow type analysis. For example, if an object has an attribute that is a decorated ndarray, you will not be able to get the fast {} accesses outside of that object:

```
cdef class item:
        def __init__(me):
                cdef ndarray<int,10> me.arr

def main():
        cdef item x = item()
        print x.arr{0}
```

```
NDArrayDecorationError
    0601_ndarray_MUST_FAIL.px,7
    ----------------------------------------------------
          5    : def main():
          6    :         cdef item x = item()
     >>>>> 7    :         print x.arr{0}
    ----------------------------------------------------
    Access to unknown decorated ndarray "x.arr".
    Known decorated ndarrays in scope: [].
```

Using `%decorate` you can do:

```
cdef class item:
        def __init__(me):
                cdef ndarray<int,10> me.arr

def main():
        cdef item x = item()
        %decorate(x.arr, ndarray<int 1d>)
        print x.arr{0}
```

```
out
out 0
```

You must do the same to access a decorated ndarray attribute of a base class:

```
cdef class base:
        def __init__(me):
                cdef ndarray<int,10> me.arr

cdef class item(base):
        def func(me):
                %decorate(me.arr, ndarray<int 1d>)
                print me.arr{0}

item().func()
```

```
out
out 0
```

Note that you can only use `%decorate` with a typed, cdef'd ndarray (see B.2.4 for discussion about typed vs untyped objects).

8.2.2 Fast creation of NDArrays

Creating ndarrays using the numpy API is slow, the creation calls, like `numpy.zeros()`, have to plumb through the Python runtime. Pex provides a faster alternative:

```
ndarray_zeros1(d1, type)
ndarray_zeros2(d1, d2, type)
ndarray_zeros3(d1, d2, d3, type)
ndarray_zeros4(d1, d2, d3, d4, type)
ndarray_zeros5(d1, d2, d3, d4, d5, type)
ndarray_zeros6(d1, d2, d3, d4, d5, d6, type)
ndarray_zeros7(d1, d2, d3, d4, d5, d6, d7, type)
ndarray_zeros8(d1, d2, d3, d4, d5, d6, d7, d8, type)
```

where `type` is one of the decoration types `char,short,int,double`, or `object`. Here is an example:

```
arr = ndarray_zeros3(17,10,5,'double')
```

The allocating declarations, e.g. `cdef ndarray<double,(17,10,5)> arr`, call these functions. For something even faster, there are also these calls

```
ndarray_empty1(d1, type)
ndarray_empty2(d1, d2, type)
ndarray_emtpy3(d1, d2, d3, type)
ndarray_emtpy4(d1, d2, d3, d4, type)
ndarray_empty5(d1, d2, d3, d4, d5, type)
ndarray_empty6(d1, d2, d3, d4, d5, d6, type)
ndarray_empty7(d1, d2, d3, d4, d5, d6, d7, type)
ndarray_empty8(d1, d2, d3, d4, d5, d6, d7, d8, type)
```

They have the same API as `ndarray_zeros` calls, but do not zero out the contents of the allocated ndarray, returning the memory as is. Thus the allocating declaration:

```
cdef ndarray<double, 10> arr
```

is equivalent to

```
cdef ndarray<double 1d> arr = ndarray_zeros1(10,'double')
```

and is slower than

```
cdef ndarray<double 1d> arr = ndarray_empty1(10,'double')
```

For arrays of dimensionality bigger than 8, you are left with calling the Python numpy API.

8.2.3 Fast Decorated Slices

Slicing ndarrays – `arr[4:7]` – is slow. Things have to plumb through the Python runtime to get to the fast numpy code that creates the slice (note that an ndarray slice is not a copy, but a "view" into the parent array). Pex provides fast, "decorated" slices – `arr{4:7}`. The general idea is that for each dimension you specify a `start:stop`. Either `start` or `stop` can be negative, if so it is counted off the end. Either can be missing, if `start` is missing it is assumed to be zero, if `stop` is missing it is assumed to `n`, where `n` is length of the dimension. Either can go off the ends of the array, and is clipped to be inside [0,n]. These semantics come from Python, see python documentation for a more complete discussion. Python slices can also have a 3rd argument – `stride` – but this is not supported for decorated slices. Here are some examples:

```
def main():
```

```
        cdef ndarray<int, 10> arr
        cdef int                      i

        for 0<=i<10: arr{i} = i

        print "  arr\n",arr
        print "\n  {3:7}\n",arr{3:7}
        print "\n  {:7}\n",arr{:7}
        print "\n  {:}\n",arr{:}
        print "\n  {-7:}\n",arr{-7:}
        print "\n  {-7:-2}\n",arr{-7:-2}
```

```
out
out   arr
out [0 1 2 3 4 5 6 7 8 9]
out
out   {3:7}
out [3 4 5 6]
out
out   {:7}
out [0 1 2 3 4 5 6]
out
out   {:}
out [0 1 2 3 4 5 6 7 8 9]
out
out   {-7:}
out [3 4 5 6 7 8 9]
out
out   {-7:-2}
out [3 4 5 6 7]
```

similarly for multiple dimensions:

```
def main():
        cdef ndarray<object,(2,4)> arr
        cdef int                             i

        for 0<=i<4: arr[0,i] = 'a' + str(i)
        for 0<=i<4: arr[1,i] = 'b' + str(i)

        print "  arr\n",arr
        print "\n  {0:1,2:4}\n",arr{0:1,2:4}
        print "\n  {:1,2:}\n",arr{:1,2:}
        print "\n  {:,:}\n",arr{:,:}
        print "\n  {1,0:1}\n",arr{1,0:1}
        print "\n  {0,-2:}\n",arr{0,-2:}
        print "\n  {:,2}\n",arr{:,2}
```

```
out
out   arr
```

```
out [[a0 a1 a2 a3]
out  [b0 b1 b2 b3]]
out
out   {0:1,2:4}
out [[a2 a3]]
out
out   {:1,2:}
out [[a2 a3]]
out
out   {:,:}
out [[a0 a1 a2 a3]
out  [b0 b1 b2 b3]]
out
out   {1,0:1}
out [b0]
out
out   {0,-2:}
out [a2 a3]
out
out   {:,2}
out [a2 b2]
```

The following example demonstrates a point of usual confusion: you may slice out the same number of elements, but depending on whether you take a number or a range of 1 number, the resulting slice will differ in the number of dimensions:

```
def main():
        cdef ndarray<object,(2,4)> arr
        cdef int                           i

        for 0<=i<4: arr[0,i] = 'a' + str(i)
        for 0<=i<4: arr[1,i] = 'b' + str(i)

        print "  arr\n",arr

        print "\n  {0,0:4}\n",  arr{0,  0:4}
        print "\n  {0:1,0:4}\n",arr{0:1,0:4}

        print "\n  {0:2,1}\n",  arr{0:2,1}
        print "\n  {0:2,1:2}\n",arr{0:2,1:2}
```

```
out
out   arr
out [[a0 a1 a2 a3]
out  [b0 b1 b2 b3]]
out
out   {0,0:4}
out [a0 a1 a2 a3]
out
```

```
out   {0:1,0:4}
out [[a0 a1 a2 a3]]
out
out   {0:2,1}
out [a1 b1]
out
out   {0:2,1:2}
out [[a1]
out  [b1]]
```

8.2.4 The `object` type

One of the supported types for decorated ndarrays is `object`, it is different from the other types `char,short,int,double`. The others are primitive C types, whereas each `object` is a Python object, and thus has an associated reference count, which is how garbage collection works in Python. Pex does the right thing for the reference count as objects are read and written to the array.

8.2.5 The `char` type

Here is an example on how to go back and forth between python strings and decorated ndarrays of characters. You may wish to do this, for example, to send strings quickly over sockets or to files using fastio (see 9.2.7):

```
import numpy,copy

def main():
    cdef ndarray<char 1d> arr

    string = "hello world"

    arr = copy.copy(numpy.frombuffer(string, dtype='b'))
    print arr
    print arr{7}
    c_printf("%c\n",arr{7})

    s = arr.tostring()
    print s
```

```
out
out [104 101 108 108 111  32 119 111 114 108 100]
out 111
out o
out hello world
```

Note the `copy.copy` call above, it is necessary because `numpy.frombuffer` returns a reference to the immutable python string, and a decorated ndarray must be writeable.

8.2.6 Type Checking

When you assign to a decorated ndarray, or when you create one by decorating an argument to a function, the operation is type checked at runtime to make sure an ndarray of the right type and dimensionality is coming in. In the following example we attempt to assign an ndarray of an incorrect type:

```
1 def main():
2         cdef ndarray<double 1d> arr
3         arr = ndarray_empty1(10,'int')
```

```
out Traceback (most recent call last):
out   File "pex", line 236, in ?
out     exit_code=module.main()
out   File "main.pyx", line 236, in main.main
out     ...
out NDArrayDecorationError: Type Mismatch: unexpected typecode for ndarray 'arr'
out         declared type: 'double'
out         expected typecode: 'd'
out         actual typecode: 'l'
```

The performance penalty for this type checking is not so bad, but if you are concerned about it, you may turn it off with a pragma:

PRAGMA	PURPOSE	DEFAULT
`pragma_ndarray_type_check`†	On/off flag for typechecking of decorated ndarrays	`True`

†set with `%whencompiling: scope.pragma_ndarray_type_check = [True | False]`

```
1 %whencompiling: scope.pragma_ndarray_type_check = False
2
3 def main():
4         cdef ndarray<double 1d> arr
5         arr = ndarray_empty1(10,'int')
```

:-(If you wish to turn off typechecking of function arguments, be sure to put the pragma before the function prototype:

```
%whencompiling:
        scope.pragma_ndarray_type_check = False

cdef func(ndarray<double 1d> arr): pass

%whencompiling:
        scope.pragma_ndarray_type_check = True
```

Were you to put the pragma in the function body, checks turn off for assignments inside the body, but not for function arguments.

8.2.7 Bounds Checking

PRAGMA	PURPOSE	DEFAULT
`pragma_ndarray_bounds_checks`†	On/off flag for bounds checking of {} accesses to decorated ndarrays	`False`

†set with `%whencompiling: scope.pragma_ndarray_bounds_checks = [True | False]`

Normal {} accesses to decorated ndarrays are not bounds checked - hence their speed. They are just like C array accesses, you may read off the end of the array and thus get garbage, or write off the end of the array and thus corrupt memory. Pex allows you to turn on bounds checking, all the {} then become slow (a 50x50 matrix multiply runs 20 times slower with bounds checks on), but safe, the array bounds are checked for every access. Turning on bounds checking is probably the first thing to try if your program coredumps.

Here we access off the end of the array, and without bounds checking, get no errors:

```
def main():
    cdef ndarray<double,10> arr
    x=arr{11}
```

and here is the same code with bounds checking turned on:

```
%whencompiling:
    scope.pragma_ndarray_bounds_checks = True

def main():
    cdef ndarray<double,10> arr
    x=arr{11}
```

```
out Traceback (most recent call last):
out   File "/res/home/dg/git/plat/pex/pex", line 334, in <module>
out        332 |
out        333 | tr(1,"calling %s.main()"%name)
out     >> 334 | exit_code=module.main()
out   File "0611_ndarray_MUST_FAIL.px", line 6, in 0611_ndarray_MUST_FAIL.main
out          4 | def main():
out          5 |     cdef ndarray<double,10> arr
out     >>   6 |     x=arr{11}
out <type 'exceptions.IndexError'>: Out of bounds index access "11"==11 for dimension 1 of "arr" which has length 10
```

There are two ways to turn on bounds checking: using the pragma, as shown above, or passing `-b` to the Pex command line. If you use the `-b` flag, first do a `pex -clean` and then run with `pex -b`. This will ensure that all of the modules used in the running program are compiled with bounds checking.

:-(If you wish to turn on bounds checking, all assignments to elements of decorated ndarrays must appear as the first thing on a line, do this:

```
%whencompiling: scope.pragma_ndarray_bounds_checks = True
if 1:
    arr{i} = 7
```

instead of this:

```
%whencompiling: scope.pragma_ndarray_bounds_checks = True
if 1: arr{i} = 7
```

8.2.8 Gotchas

:-(To get the lengths of an ndarray, use the builtin `dimlen(arr,dimension)` function instead of `.shape` attribute, do this: SLOW

```
cdef int i,j
for 0<=i<dimlen(arr,0):
    for 0<=j<dimlen(arr,1):
        pass
```

not this:

```
cdef int i,j
for 0<=i<arr.shape[0]:
    for 0<=j<arr.shape[1]:
        pass
```

`.shape` is a Python tuple, getting it as an attribute of an ndarray is slow, and accessing its elements is slow, whereas `dimlen()` is a C function, and thus runs at C speed.

:-(Do not use negative indices with {} accesses. If you do, you'll access garbage memory from before the beginning of the ndarray, same as would happen with a C array. Turning on bounds checks (see 8.2.7) will detect this error. DEATH!

:-(Can not have global decorated ndarrays.

:-(Can not do

```
tup = (3,4)
cdef ndarray<double,tup> arr
```

must do

```
tup = (3,4)
cdef ndarray<double,(tup[0],tup[1])> arr
```

:-(If you have a decorated ndarray as an attribute of a cdef class, you can not access it with {} outside of the class, or even in subclasses. This is a result of Pex's lack of an honest type system. See 8.2.1 for how to overcome this limitation with `%decorate`.

:-(Can not use {} accesses inside declarations, e.g. can not do `cdef int x=arr{i}`.

:-(You must make sure index variables you use with {} accesses are cdef'd ints, otherwise things will be slow: SLOW

```
for 0<=i<dimlen(arr,0):
    func( arr{i} )
```

In the snippet above, `i` is a Python object, and gets converted to a C int for every {} access, and thus every time through the loop. Instead do:

```
cdef int i
for 0<=i<dimlen(arr,0):
    func( arr{i} )
```

:-(Use the following loop form: SLOW

```
cdef int i
for 0<=i<dimlen(arr,0):
```

```
3     func( arr{i} )
```

instead of

```
cdef int i
for i in range(dimlen(arr,0)):
    func( arr{i} )
```

Otherwise things will be slow (this is Pyrex syntax that allows you to write C fast loops).

:-(To allow the fast {}-style accesses, behind the scenes Pex unpacks certain internals of the ndarray into local variables. As a consequence, if an ndarray is re-assigned, the internals have to be unpacked again, else bad things will happen. For this reason Pex attempts to track the re-assignments, so that it can unpack the internals immediately after it happens. If Pex misses the change to the array, the local variables become stale, and every subsequent {} access corrupts memory. DEATH!

Pex is able to track simple assignments, such as:

```
arr = x
```

and compound assignments, such as:

```
arr, arr2 = x, y
```

Any other way of assigning to a decorated ndarray variable will cause a memory corruption.

For example,

```
cdef ndarray<int 1d> arr
for arr in arrs:
    arr{0}=7
```

Pex is not able to see `for arr in arrs:` as a re-assignment, and so does not appropriately unpack the new ndarray each time through the loop. You can instead do:

```
cdef ndarray<int 1d> arr
for i in range(len(arrs)):
    arr = arrs[i]
    arr{0} = 7
```

Here is another way memory can get corrupted, suppose you have a decorated ndarray that is a class attribute. From inside the class method, you call another class method which re-assigns the attribute:

```
cdef class item:
    def __init__(me):
        cdef ndarray<int,10> me.arr
    def change(me):
        me.arr = ndarray_zeros1(5,'int')
    def func(me):
        me.change()
        me.arr{3}
x=item()
x.func()
```

The above corrupts memory, you need to add `me.arr = me.arr` after `me.change()` in `func` to force the internals to be re-unpacked after the call to `change`.

If you discover that a decorated array is being re-assigned without Pex being able to track it, this is the recommended hack to fix things: assign the array to itself, as in `arr = arr`. Pex will find this assignment, and will unpack the internals of the array appropriately.

:-(Pex gives you two ways to declare class attributes (see C.1 for full discussion), in the class preamble, and inside the `__init__` method. Allocating ndarrays declarations, however, may only be declared inside the `__init__` method, therefore do this:

```
cdef class item:
    def __init__(me):
        cdef ndarray<double,10> me.arr
```

not this:

```
cdef class item:
    cdef ndarray<double,10> arr
```

Non-allocating declarations you may still put in the preamble:

```
cdef class item:
    cdef ndarray<double 1d> arr
```

9 Modest Cdef Classes

9.1 Simple

When you write a `cdef class` in Pyrex (see B.2.3 about why you may want to), a lot of the niceties you get for free with Python classes go away. Some of the downsides: you can't print the resulting objects, can't compare them for equality, can't access their attributes from

Python, and most importantly can't automatically write/read them from disk ("pickling" in pythonese). Pex brings a lot of these niceties back, provided your cdef class is "modest" – its attributes come from a restricted set of types.

To make yourself a modest cdef class, restrict your attributes to the following types:

(i.) `bool, char, uchar, short, ushort, int, uint, int64, uint64, float, double`: these are primitive C types (see 12 for more info)

(ii.) `object`: this is Pyrex's type name for any Python object (list, tuple, class, etc)

(iii.) `<any_other_cdef_class>`: this is `foo` if you have `cdef class foo:` defined somewhere.

Thus the following is a modest cdef class (note that you can have any python object (list, tuple, dictionary, etc) be an attribute of a cdef class with type `object`):

```
cdef class modest:
        def __init__(me):
                cdef int    me.i
                cdef object me.ob = [1, 2, 3, 4]
```

and the following is immodest:

```
cdef class immodest:
        cdef int *modesty_violating_pointer
```

If you further restrict yourself to only the primitive C types and decorated ndarrays of any simple numerical type, e.g. anything but `object` (see 8 about decoration), you get an "unspoiled" class, which gets all the niceties of a modest class, and also `fastio` (see 9.2.7) – a serialization method that is about 10-12 times faster than pickling. Here is an unspoiled class:

```
cdef class unspoiled:
        def __init__(me):
                cdef int                    me.i
                cdef ndarray<char,(10,10)> me.arr
```

and here is one that, while modest, is not unspoiled:

```
cdef class modest_but_not_unspoiled:
        def __init__(me):
                cdef ndarray me.arr
```

This class is spoiled because the ndarray `me.arr` is not decorated.

9.2 Elaborate

9.2.1 One Pragma to Rule Them All

Each of the niceties described below, has its own pragma switch, but there is also one pragma that turns off all of the automatically generated methods described in this section:

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_all_off`†	On/off flag to control the generation of all convenience methods for modest and unspoiled classes	`True`

†set with `%whencompiling: scope.pragma_gen_all_off = [True | False]`

9.2.2 `__str__()` – conversion to a string

`cdef class` requirements: modesty

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_strmeth`†	On/off flag to control the generation of the `__str__()` method	`True`

†set with `%whencompiling: scope.pragma_gen_strmeth = [True | False]`

The `__str__()` method is called by Python when you attempt to print an object, or convert it to a string. If it is absent, a less exciting string is produced, showing the object's type and memory address. For example, this is what you get when you turn off `__str__()` method generation:

```
%whencompiling: scope.pragma_gen_strmeth=False

cdef class item:
        cdef int x

print item()
```

```
out
out <0804_strmeth.item object at 0xa19ac2c>
```

If you leave it in, you get:

```
cdef class item:
        cdef int x

print item()
```

```
out
out c{x = 0}
```

One possible reason to turn it off, is if you intend to write a custom `__str__()` method.

9.2.3 `_equal_()`

`cdef class` requirements: modesty

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_equalmeth`†	On/off flag to control the generation of the `_equal_()` method	`True`

†set with `%whencompiling: scope.pragma_gen_equalmeth = [True | False]`

This method allows you to compare two modest `cdef` classes:

```
cdef class item:
        cdef int i

cdef item x=item(), y=item()

print x._equal_(y)
y.i=7
print x._equal_(y)
```

```
out
out 1
out 0
```

Gotchas

:-(Unless all of your attributes are primitive C types, the `_equal_()` will happen at Python speeds. In principle, for attributes that are `cdef` classes or ndarrays it could be C fast, but at the moment isn't. SLOW

:-(This is a `cdef` method, and so not callable from Python directly. However you can access it through `==` and `!=`, see next section.

9.2.4 `__richcmp__()` – Python comparisons

`cdef class` requirements: modesty

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_richcmpmeth`†	On/off flag for the generation of the `__richcmp__()` method	`True`

†set with `%whencompiling: scope.pragma_gen_richcmpmeth = [True | False]`

When Pyrex sees a `==` or `!=` comparison between two `cdef` classes, it calls the `__richcmp__()` method of one of the objects. If this method is absent, the two objects are compared based on their memory address, thus even if two instances have every attribute the same, they will never be equal:

```
%whencompiling: scope.pragma_gen_richcmpmeth = False

cdef class item:
        cdef int i

cdef item x=item(), y=item()
print x,y
print x==y
```

```
out
out c{i = 0} c{i = 0}
out False
```

Pex generates this method to call the generated `_equal_()` method (see previous section), and this causes the `==` and `!=` checks to happen based on the actual contents of the two objects, not just their memory addresses:

```
cdef class item:
        cdef int i

cdef item x=item(), y=item()
print x,y
print x==y
```

```
out
out c{i = 0} c{i = 0}
out True
```

Gotchas

:-(These `==` and `!=` checks are slow because they plumb through Python. SLOW

9.2.5 `_todict_()`, `_fromdict_()` – dict coercion

`cdef class` requirements: modesty

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_dictcoercion`†	On/off flag for the generation of the `_todict_()` and `_fromdict_()` dictionary coercion methods	`True`

†set with `%whencompiling:` `scope.pragma_gen_dictcoercion = [True | False]`

Attributes of `cdef` classes are not visible from Python. This is illustrated by the following example where we are not able to access the attribute `i` from `main.py`:

```
mod.px
        cdef class item:
                cdef int i

main.py
        import pex
        mod = pex.pimport('mod')
        ob = mod.item()
        print ob.i
```

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    print ob.i
AttributeError: 'mod.item' object has no attribute 'i'
```

To overcome this limitation, Pex generates two methods for your `cdef` classes: `_todict_()` returns the `cdef` class' attributes in a dictionary, `_fromdict_()` sets them from a dictionary:

```
mod.px
        cdef class item:
                cdef int i

main.py
        import pex
        mod = pex.pimport('mod')

        ob = mod.item()
        print 'ob  ',ob

        d = ob._todict_()
        print 'dict',d
```

```
14
15          ob._fromdict_({'i': 17})
16          print 'ob  ',ob
```

```
out
out ob   c{i = 0}
out dict {'i': 0}
out ob   c{i = 17}
```

Gotchas

:-(Any extra keys in the dictionary that is passed into `_fromdict_()` are ignored. For example `ob._fromdict_({'i': 17, 'rubberchicken': True})` would have worked just as well in the above example.

9.2.6 `__reduce__()`, `__setstate__()` – Python pickling

`cdef class` requirements: modesty

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_pickle`†	On/off flag for the generation of the `__reduce__()` and `__setstate__()` pickling methods	`True`

†set with `%whencompiling: scope.pragma_gen_pickle = [True | False]`

Normal Pyrex `cdef` classes are not "picklable" (pickling is Python's term for serialization – writing and reading objects from disk). Pex automatically generates two special Python methods `__reduce__()` and `__setstate__()`, which make your `cdef` classes picklable. It is unlikely you'll ever need to call these functions directly.

A thing of note is that the Python's object `copy` machinery also works through these methods, and so their presence makes your `cdef` classes copiable:

```
1 import copy
2
3 cdef class item:
4         cdef int i
5
6 cdef item x = item()
7 x.i = 3
8 y   = copy.copy(x)
```

See the next section for a faster serialization method, `fastio`. Pickling uses fastio when possible to make things run faster, so its generation must be turned on for pickling to work.

Performance

The main thing that determines the speed of pickling for a cdef class is the type of its attributes. Given an attribute, if it is of a simple C type (`bool, char, uchar, short, ushort, int, uint, int64, uint64, float, double`), it can be written directly at C speed. If it is of type `object`, or any other type other than decorated ndarray (including another cdef class, or undecorated `ndarray`), it must go through the standard python pickling machinery and is written "pythonically", which is usually significantly slower. Decorated ndarrays split into several cases. Decorated ndarrays of objects are always written pythonically. Decorated ndarrays of any other type (`char, short, int, double`), can be written direct or pythonically, depending on the number of objects that refer to them. Given such a decorated ndarray, if only the object whose attribute it is has a reference to the ndarray, it will be written direct, if more than one, it will go pythonically. For example, in the following code, `arr` will be written direct,

```
1  import pex.pxpickle
2
3  cdef class item:
4     cdef ndarray<char 1d> arr
5
6  def main():
7     cdef item i
8     i = item()
9     i.arr = ndarray_zeros1(5,'char')
10    pex.pxpickle.dumps(i)
```

but in the following, it will be written pythonically:

```
1  import pex.pxpickle
2
3  cdef class item:
4     cdef ndarray<char 1d> arr
5
6  def main():
7     cdef item i = item()
8     i.arr = ndarray_zeros1(5,'char')
9     obj = i.arr      # this adds another reference on i.arr
10    pex.pxpickle.dumps(i)
```

In the above example, the array "i.arr" is also referred to by "obj", two references.

The number of references to objects is also what determines the amount of memory the pickling process uses. Roughly speaking, any object with more than one reference on it needs

to be "memoized" by the pickler, which costs approximately 70 bytes of memory. This can be significant, if you have 10 million objects with more than 1 reference, you'll need roughly 700MB of memory to pickle them out. Sometimes even objects with 1 reference on them get memoized, this usually happens because the pickling process creates temporary objects which create extra references on your objects, and these extra references cause them to be memoized (unnecessarily, but it still costs memory).

In order to avoid having objects memoized because of these temporary references, pex comes with a slightly modified `cPickle` module, called `pex.pxpickle`. It is fully compatible with the usual python pickle format, and has the same API as `cPickle`. When you use it in conjunction with lots of `cdef class` objects pickling will use significantly less memory overhead than if you use `pickle` or `cPickle`, so the recommendation is to use it in all cases. You may also want to turn off the garbage collector when pickling things out, as this may speed up running time around 30%. Given below is the suggested way to pickle out your objects:

```
import pex.pxpickle,gc
b = <some object of yours>
gc.disable()
pex.pxpickle.dumps(b)
gc.enable()
gc.collect()
```

The following table summarizes how various attributes of cdef classes are written, and how and if they are memoized. Below, `R` refers to the reference count – the number of references on an object. A numpy array is considered to be well-behaved if `arr.flags & (NPY_C_CONTIGUOUS | NPY_OWNDATA | NPY_WRITEABLE | NPY_ALIGNED)`.

pythonic write	**Attribute Type**	memoized
never	SIMPLE C TYPE	never
always	`object`	if `R>1`
if `R>1` or array not well-behaved	`ndarray<{char|short|int|double}>`	if `R>1`
always	`ndarray<object>`	if `R>1`
always	`object` in `ndarray<object>`	if `R>1` or parent ndarray not well-behaved or parent ndarray has `R>1`

The above only refers to objects that are attributes of cdef classes, and only when the

`pex.pxpickle` module is used. If you have, say, a list of objects, or use `pickle` or `cPickle` these rules do not apply.

If you are wondering about the exact set of objects memoized by the pickler, you can examine it like so:

```
import pex.pxpickle
b = <some object of yours>
fd = StringIO.StringIO()
p = pex.pxpickle.Pickler(fd)
p.dump(b)
memoized_objects = [tup[1] for tup in p.memo.values()]
```

Gotchas

:-(PICKLING OUT CDEF CLASSES IS NOT A THREAD SAFE OPERATION!

:-(If you have a list of `cdef class` objects you wish to pickle, do not dump them one at a time like so:

```
import pex.pxpickle

cdef class item:
        cdef int i

list = [item() for i in range(100)]

file = open('somefile','w')
for ob in list: pex.pxpickle.dump(ob,file)
file.close()
```

Instead do the whole list at once:

```
import pex.pxpickle

cdef class item:
        cdef int i

list = [some_cdef_class() for i in range(100)]

file = open('somefile','w')
pex.pxpickle.dump(list,file)
file.close()
```

Pex dumps a type signature once for every dump invocation, thus once for every object when you write them one at a time. This unnecessarily, and possibly substantially, increases your

storage requirements. In the second case, when you write the entire list at once, the type signature is written only once.

9.2.7 `_fastload_()`, `_fastdump_()`, `_memload_()`, `_memdump_()` – fastio

`cdef class` requirements: must be unspoiled

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_fastio`†	On/off flag for the generation of the `_fastdump_()`, `_fastload_()`, `_memload_()`, and `_memdump_()` fastio methods	`True`

†set with `%whencompiling: scope.pragma_gen_fastio = [True | False]`

If your `cdef class` is unspoiled, in addition to being modest (see 9.1), Pex will generate the four methods, `_fastdump_()`, `_fastload_()`, `_memload_()`, and `_memdump_()` which implement a serialization scheme called "fastio". It is slightly less convenient to use than pickling, but is 10-12 times faster writing to files, more so writing directly to memory. `_fastdump_()`, `_fastload_()` write `cdef class` objects to files (and special file-like objects), `_memload_()`, and `_memdump_()` serialize directly to memory.

Here is an example showing basic usage when dealing directly with memory:

```
cdef class item:
    cdef int i

cdef item x = item()

cdef ndarray buf = ndarray_zeros1(10,'char')

nbytes = x._memdump_(buf.data,dimlen(buf,0))
assert nbytes == 6, "2 for checksum, 4 for the int"

cdef item y = item()

nbytes = y._memload_(buf.data,dimlen(buf,0))

# you can also use char* directly
cdef char buf2[10]
x._memdump_(buf2,10)

# if you want to just get the number of bytes the
```

```
# object would take to serialize do the following
nbytes = x._memdump_(NULL,-1)
```

The example above shows the several different ways to use these functions: using an ndarray for the buffer, using a regular char array, and just getting the size it would take to serialize an object without allocating any memory. The functions throw an `IOError` if the buffer passed in is not large enough.

Here is an example showing basic usage when writing to files, or file-like objects:

```
cdef class item:
        cdef int i

cdef item x = item()

x._fastdump_(open('tempfile','w'))

cdef item y = item()

y._fastload_(open('tempfile'))
```

One common use case is to read in several `cdef class` objects using fastio. In order to call the `_fastload_()` method, you must have an instance of the `cdef class`, but it makes no sense to create one if its contents are immediately overwritten by the `_fastload_()` call. For this reason Pex provides the `pex_create_uninitialized()` function (see 11), which instantiates a `cdef class` without calling its constructor – all attributes that are primitive C types are set to zero, all attributes that are Python objects are set to `None`. The following example illustrates this case:

```
import pickle

cdef class item:
        cdef int i

# WRITE

list = [item() for i in range(100)]

cdef item x

file = open('tempfile','w')
pickle.dump(len(list),file)
for x in list: x._fastdump_(file)
file.close()

# READ

```

```
list = []

cdef item y

file = open('tempfile')
n    = pickle.load(file)
for i in range(n):
        y = pex_create_uninitialized(item)
        y._fastload_(file)
        list.append(y)
file.close()
```

You can also use fastio to send cdef classes over a socket:

```
datatype.px
        cdef class item:
                cdef int    i
                cdef double d

client.px
        % from datatype pimport item

        import socket

        HOST = 'localhost'
        PORT = 33838

        if 1: # setup connection
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect((HOST,PORT))

                name  = str(s)
                sfile = PyFile_FromFile(c_fdopen(s.fileno(),'w'),
                                        name, 'w', c_fclose)

        cdef item x = item()

        x._fastdump_(sfile)

server.px
        % from datatype pimport item

        import socket

        HOST = 'localhost'
        PORT = 33838

        if 1: # setup connection
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
                s.bind(('', PORT))
                s.listen(1)
                s_conn, addr = s.accept()

                name  = str(s_conn)
                sfile = PyFile_FromFile(c_fdopen(s_conn.fileno(),'r'),
                                        name, 'r', c_fclose)

        cdef item x = pex_create_uninitialized(item)

        x._fastload_(sfile)

        print 'received',x
```

In addition to taking a python file object as an argument, `_fastload_` and `_fastdump_` methods also can take an instance of a class derived from `pex.pexruntime.BaseFastIOStream`, which you may write in order to serialize to things other than files or sockets. Here is an example that allows you to directly read and write objects to zip files using zlib. This class is included in `pex.pexruntime` (see 10).

```
cdef extern from "zlib.h":
    ctypedef void* gzFile
    gzFile gzopen(char *path, char *mode) except exits
    int gzread(gzFile file, void *buf, unsigned len) except exits
    int gzwrite(gzFile file,void *buf, unsigned len) except exits
    int gzclose(gzFile file) except exits
    char* gzerror(gzFile file, int *errnum) except exits

cdef class GzFile_FastIOStream(pex.pexruntime.BaseFastIOStream):
    %whencompiling: scope.pragma_gen_all_off = True

    def __init__(me,path,mode='r'):
        cdef void *me.stream = gzopen(path, mode)

    cdef void fastwrite(me, void *data, int nbytes):
        gzwrite(me.stream,data,nbytes)

    cdef void fastread(me, void *data, int nbytes):
        gzread(me.stream,data,nbytes)

    def close(me):
        gzclose(me.stream)
        me.stream = NULL

    def __dealloc__(me):
        if me.stream <> NULL:
            me.close()

```

```
cdef class item:
    cdef int x

def main():
    cdef item x = item()

    gzfile = GzFile_FastIOStream('foo.gz','w')
    x._fastdump_(gzfile)
    gzfile.close()

    gzfile = GzFile_FastIOStream('foo.gz')
    x._fastload_(gzfile)
```

The fastio stream classes also define a `pickleload` and `pickledump` methods that can read and write any python objects to the stream. You don't need to implement these methods yourself, they are in `BaseFastIOStream` and work once you define `fastwrite` and `fastread` in the derived class.

Gotchas

:-(FASTIO/MEMDUMP IS NOT A THREAD SAFE OPERATION!

:-(These are `cdef` methods, and as such, are not visible from Python.

:-(If you have an attribute that is a decorated ndarray that has become non-contiguous in memory, the `_fastdump_()` will fail. An ndarray may become non-contiguous if it is a slice from another ndarray with a slice step bigger than 1. You may perhaps restore its contiguousness with:

```
me.arr=PyArray_ContiguousFromAny(me.arr,<NPY_TYPES>self.arr.decsr.type_num,0,0)
```

but you are really out of bounds with this one. Note that if this works, it will be a full copy of the array, and also you may only do this from Pex, not sure how to do it from Python.

:-(Here is a most natural mistake to make, and it is guaranteed to produce a coredump. It is so natural that you'll probably make it even after reading this text, DEATH!

```
cdef item y
y._fastload_(open('somefile'))
```

The above coredumps because `y`, while declared, was never instantiated (see 22 for complete discussion of this kind of an error). Before calling `y._fastload_()` you must first instantiate `y` like this `y=item()` or this `y=pex_create_uninitialized(item)`.

9.2.8 `_deepcopy_()` – fast copy

`cdef class` requirements: must be unspoiled

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_deepcopy`†	On/off flag for the generation of the `_deepcopy_()` method	`True`

†set with `%whencompiling: scope.pragma_gen_deepcopy = [True | False]`

This function provides a fast copy method for an unspoiled cdef class:

```
cdef class item:
        cdef int i
        cdef ndarray<double 1d> arr

        def __init__(self):
           self.i = 17
           self.arr = ndarray_zeros1(5,'double')
           self.arr{2} = 12

cdef item x, y

x = item()
y = pex_create_uninitialized(item)

print 'before copy',x,y

y._deepcopy_(x)

print ' after copy',y
```

```
out
out before copy c{arr = array([  0.,   0.,  12.,   0.,   0.]), i = 17} c{arr = None, i = 0}
out  after copy c{arr = array([  0.,   0.,  12.,   0.,   0.]), i = 17}
```

SLOW

If one of the attributes is an ndarray, and it is of a different size in the destination class than in the source, it will be reallocated. Since you are presumably using this copy method for speed, you should be aware of this, as the constant re-allocation can slow things down, as can thrashing if the size of the ndarray keeps changing.

9.2.9 `__hash__()` – Python hashing

`cdef claqss` requirements: none

PRAGMA	PURPOSE	DEFAULT
`pragma_gen_hashmeth`†	On/off flag for the generation of the `_hash_()` method	`True`

†set with `%whencompiling: scope.pragma_gen_hashmeth = [True | False]`

This function returns the memory address of the object. This function is needed because if an object has a `__richcmp__()` method, but not a `__hash__()` method, Python will not allow the object to serve as a key to a hash:

```
%whencompiling: scope.pragma_gen_hashmeth = False

cdef class item:
        cdef int i

d = {}

x = item()

d[x] = 'foo'
```

```
Traceback (most recent call last):
  <-----------------<snip>-------------------<snip>------------------>
  File "0814_hash_MUST_FAIL.pyx", line 168, in 0814_hash_MUST_FAIL
    d[x] = 'foo'   ## 0814_hash_MUST_FAIL.px,10
TypeError: unhashable type
```

Gotchas

:-(This is not a real hash, it's just the memory address of the object.

9.2.10 `_typesig_()`

`cdef class` requirements: none

You'll probably never need to call this function directly, it is used by the pickling and fastio machinery. Here it is in any case, it gives the C type signatures of all the attributes:

```
cdef class item:
        def __init__(self):
                cdef int                 self.i
                cdef object              self.ob
                cdef ndarray<double 7d> self.arr
```

```
6
7 print item()._typesig_()
```

```
out
out (('int', 'i'), ('ndarray', 'arr', 'double', 7), ('object', 'ob'))
```

9.2.11 How Things Can Go Wrong

Some of the methods described above depend on each other, for example `__str__()` on `_todict_()`. If you run Pex with the `-W` command line flag, you will get warnings if you've turned off some method, but not the methods that depend on it:

```
1 %whencompiling:
2         scope.pragma_gen_dictcoercion = False
3
4 cdef class item:
5         cdef int i
```

```
$ pex -W main.px
[WARNING] main.px,4: __str__ method of cdef class "item" will not work,
        it depends on _todict_ method, whose generation was turned off with a
        pragma
[WARNING] main.px,4: pickling of cdef class "item" will not work, it
        depends on dict coercion methods, whose generation was turned off with a
        pragma
```

The more usual way things can go wrong is if you make your `cdef class` immodest, for example by giving it an attribute that is a pointer. This will not cause a compile time error – all the methods will still be generated – but they'll just be stubs that throw a `PexNotGeneratedException`:

```
1 cdef class item:
2         cdef int *modesty_violation_pointer
3
4 item()._todict_()
```

```
out PexNotGeneratedError
out
out         Method not auto-generated by pex because this class has attributes that pex
out         doesn't know how to convert to python objects. Problem attributes:
out
out             int *modesty_violation_pointer
out
```

```
out         Possible solutions: you may change this class to only use supported attribute
out         types (which still may not work for some problems):
out
out             bool, char, uchar, short, ushort, int, uint, int64, uint64, float, double,
out             cdef classes, python objects
out
out         or turn off all pex processing with
out
out             %whencompiling: scope.pragma_gen_dictcoercion = False
out
out         and then implement this method yourself.
out
```

Recall that for a modest class, an attribute, among other things, can be another `cdef class`. When Pex sees an attribute declaration `cdef <sometype> attr` and `<sometype>` isn't a primitive C type or `object`, Pex assumes it is another `cdef class`. In fact, it could be something else, a `struct` for example, but Pex can't figure this out because it does not perform a deep enough level of analysis. This case leads to cryptic compile time errors, like so:

```
1 cdef extern from "stdio.h":
2         cdef struct FILE:
3                 pass
4
5 cdef class item:
6         cdef FILE f
```

```
out PyrexError
out     python cython.py main.pyx
out
out     Error converting Pyrex file to C:
out     ------------------------------------------------------------
out             def _todict_(me):                   ## made by pex
out               d = {
out                 'f': me.f,
out                       ^
out     ------------------------------------------------------------
out     main.pyx:78:19: Cannot convert 'FILE' to Python object
out
out     Error converting Pyrex file to C:
out     ------------------------------------------------------------
out             def _fromdict_(me,dict):            ## made by pex
out               me.f = dict['f']
out                         ^
out     ------------------------------------------------------------
out     main.pyx:83:21: Cannot convert Python object to 'FILE'
out
```

```
out     Error converting Pyrex file to C:
out     ------------------------------------------------------------
out               cdef bool equal=cTrue
out
out               if   me.f <> other.f: equal = cFalse
out                         ^
out     ------------------------------------------------------------
out     main.pyx:97:20: Invalid types for '!=' (FILE, FILE)
```

9.2.12 How to Write Your Own

If you wish to write your own implementation for any of the methods above, turn the method(s) off with a pragma. For pickling or fastio, you'll probably want to look at Pex generated code first, to get an idea of what the method is supposed to do.

10 pex.pexruntime

This is a module that is available to all pex modules by default, and includes certain convenient cdef classes and methods. For the moment it has an abstract base class for fastio serialization, as well as one concrete implementation that allows serialization to zip files, see 9.2.7 for more information.

11 pex_create_uninitialized()

Sometimes, as in the case of fastio (see 9.2.7), you may wish to quickly instantiate your `cdef class foo:`, without going through the constructor. You may do so using `pex_create_uninitialized(foo)`, like so:

```
1 cdef class item:
2         cdef int     i
3         cdef char    c
4         cdef object  h
5         cdef ndarray arr
6
7         def __init__(me): print "Woe! I shall never be called"
8
9 print pex_create_uninitialized(item)
```

```
out
out c{arr = None, c = 0, h = None, i = 0}
```

All attributes that are primitive C types are set to zero, all attributes that are Python objects are set to `None`.

Gotchas

:-(Though technically `pex_create_uninitialized()` may take any type object as its one argument, do not call it with anything other than a Pyrex `cdef class`, e.g. don't do this:

```
class item: pass
pex_create_uninitialized(item)
```

Bad things may happen.

12 Primitive C Types and Literals

Pex supports the following primitive C types:

`bool`	1 byte	+
`char`, `uchar`	1 byte	+
`short`, `ushort`	2 bytes	++
`int`, `uint`	4 bytes	++++
`int64`, `uint64`	8 bytes	++++++++
`float`	4 bytes	++++
`double`	8 bytes	++++++++

Here is an example where we print the size of the type and the maximum value:

```
                # MAX VALUE
cdef bool   b    = cTrue
cdef char   c    = 0x7f
cdef uchar  uc   = 0xff
cdef short  s    = 0x7fff
cdef ushort us   = 0xffff
cdef int    i    = 0x7fffffff
cdef uint   ui   = 0xffffffff
cdef int64  i64  = 0x7fffffffffffffffL
cdef uint64 ui64 = 0xffffffffffffffffUL
cdef float  f    = 3.40282346638528860e+38
cdef double d    = 1.79769313486231570e+308

print 'bool',   sizeof(bool),   b
print 'char',   sizeof(char),   c
print 'uchar',  sizeof(uchar),  uc
print 'short',  sizeof(short),  s
print 'ushort', sizeof(ushort), us
```

```
19 print 'int',    sizeof(int),    i
20 print 'uint',   sizeof(uint),   ui
21 print 'int64',  sizeof(int64),  i64
22 print 'uint64', sizeof(uint64), ui64
23 print 'float',  sizeof(float),  f
24 print 'double', sizeof(double), d
```

```
out
out bool 1 1
out char 1 127
out uchar 1 255
out short 2 32767
out ushort 2 65535
out int 4 2147483647
out uint 4 4294967295
out int64 8 9223372036854775807
out uint64 8 18446744073709551615
out float 4 3.40282346639e+38
out double 8 1.79769313486e+308
```

All variables of these types are pure C variables, all arithmetic and boolean expressions with them are compiled directly to C, and so run at C speed, but see 21 for situations when such variables are upcast to Python and so become slow.

In the example above, we added the suffix "L" to the constant we assigned to `i64`. Were we to do this:

```
1 cdef int64 i64 = 0x7fffffffffffff
```

the C compiler would complain with something like:

```
warning: integer constant is too large for ‘‘long’’ type
```

The C compiler does not like integer constants (also known as "literals") that are bigger than what can fit into 4 bytes. To appease it, we append the "L":

```
1 cdef int64 i64 = 0x7fffffffffffffL
```

An unsigned int can be bigger than a signed int, so when assigning something big to a `uint64`, append a "UL":

```
1 cdef uint64 ui64 = 0xffffffffffffffUL
```

Now, lets say you are ready to get serious, and want to push some big numbers around, numbers even bigger than 8 bytes. Python supports arbitrary precision integers, to specify such a literal, add the suffix "pyL" to your number:

```
x = 191561942608236107294793378084303638130997321548169216pyL
```

which will get translated to the C code:

```
PyLong_FromString("191561942608236107294793378084303638130997321548169216",0,0)
```

All operations on such numbers will be much slower than operations on primitive C types.

Gotchas

:-(Do not confuse these types with the ones available for ndarray decoration (see 8 for those).

:-(If you do the following:

```
x = 3
cdef uchar c
c = x
```

```
out 1005_uchar_and_ushort.c: In function init1005_uchar_and_ushort:
out 1005_uchar_and_ushort.c:1375: warning: comparison is always false due to limited range of data type* pex.build  c
```

you get a strange compiler error. This happens because behind the scenes your code is translated to the following C code (roughly):

```
PyObject *x = PyInt_FromLong(3);
uchar c = PyInt_AsLong(x); if (c==-1) { <some error handling> }
```

Of course `c` can never be less than zero, being unsigned and all, hence the compiler warning. This is a Pyrex limitation, heed your compiler warning and code around this somehow, for example assign first to an `int` and if that succeeds assign to your `unsigned` variable. Same problem exists for `ushort`, but due to Pyrex's internal architecture, not for `uint` or `uint64`.

13 C `struct` and `typedef`

Pyrex supports C `struct` and `typedef`, Pex support is more limited. You may use them, just as in Pyrex, as global or local variables, but you may not have them as function arguments for `cdef` functions or as `cdef class` attributes. If you need to pass them around, do so as `void` pointers and then cast them locally to the right type. Here is an example:

```
cdef extern from "time.h":

```

```
3       ctypedef int time_t
4
5       cdef struct tm:
6           int tm_sec, tm_min, tm_hour, tm_mday, tm_mon
7           int tm_year, tm_wday, tm_yday, tm_isdst
8
9       time_t time(time_t*) except exits
10      tm *localtime(time_t*) except exits
11
12  cdef class item:
13
14      cdef void* vt
15
16      def __init__(self):
17          self.vt = c_malloc(sizeof(tm))
18
19      def save_time(self):
20          cdef tm *s, *t
21          cdef time_t x
22
23          x = time(NULL)
24          s = localtime(&x)
25
26          t = <tm*>self.vt
27
28          c_memcpy(t, s, sizeof(tm))
29
30      def day_of_year(self):
31          cdef tm *t
32
33          t = <tm*>self.vt
34
35          return t[0].tm_yday
36
37      def __dealloc__(self):
38          if self.vt <> NULL:
39              c_free(self.vt)
40
41  def main():
42      x = item()
43      x.save_time()
44      print x.day_of_year()
```

```
out
out 69
```

Note that if you `malloc` in the contructor, as in the example above, you must make a special `__dealloc__` method which is called at object destruction, and put in the correponding `free`,

else you'll leak memory. Also note that because you have a `void*` as a `cdef class` attribute, you lose all of the niceties described in Section 9, such as the ability to pickle or convert instances of the class to strings.

14 Configuring Compilation

Using the `%whencompiling` directive, you can change the flags passed to the C compiler or linker, set pragmas, and in general execute arbitrary Python code at the time your `.px` file is compiled into a `.so` file. For example:

```
%whencompiling: print "hello world"
```

ran with

```
$ pex main.px
hello world
```

causes Pex to execute your `print` statement while compiling this `main.px` into a `main.so`. If you run Pex again, the print will not be executed:

```
$ pex main.px
$
```

because `main.so` was already compiled.

Pex gives you access to two objects in your `%whencompiling` code, `env` and `scope`; here is what they contain:

```
%whencompiling:
        from pprint import pprint

        print 'ENV'
        pprint(env.__dict__)
        print

        print 'SCOPE'
        pprint(scope.__slots__)
        print
```

```
out ENV
out {'cc': ['gcc',
out         '-c',
out         '-fPIC',
out         '-O6',
```

```
out         '-fno-strict-aliasing',
out         '-I/usr/local/include/python2.6',
out         '-I/res/home/dg/git/plat/pex',
out         '-I/opt/lib/python2.6/site-packages/numpy/core/include'],
out  'is_pex_runtime_module': False,
out  'link': ['gcc', '-shared', '-lm', '-lz'],
out  'n_preamble_lines': 11,
out  'pex_file': '1100_whencompiling.px',
out  'pimports': ['pex.pexruntime'],
out  'pimports_abspath': ['/res/home/dg/git/plat/pex/pexruntime.px'],
out  'print_backtrace_on_segfault': True,
out  'print_warnings': False,
out  'pxd_file_content': 'include "numpy.pxi"\ninclude "px_ndarray.pxi"\n\ninclude "builtin.pxi"\ncimport pex.pexrunt
out  'pxd_file_indent': 0,
out  'pyrex': ['python /res/home/dg/git/plat/3rdparty/Cython_Bundle/cython/cython.py',
out            '-I/res/home/dg/git/plat/pex'],
out  'used_ndarray_decoration': False,
out  'whencompiling_directive_globals': {'env': <pex_base.struct instance at 0x9fe6d4c>,
out                                      'pprint': <function pprint at 0xb7c58a04>,
out                                      'scope': <px_to_pyx.Scope object at 0x9fead94>}}
out
out SCOPE
out ['depth',
out  'parent',
out  'ndarrays',
out  'line_for_declarations',
out  'line',
out  'self_arg',
out  'func_name',
out  'variables',
out  'non_cdef_class_types',
out  'pragma_ndarray_bounds_checks',
out  'pragma_ndarray_type_check',
out  'pragma_gen_dictcoercion',
out  'pragma_gen_strmeth',
out  'pragma_gen_equalmeth',
out  'pragma_gen_richcmpmeth',
out  'pragma_gen_hashmeth',
out  'pragma_gen_pickle',
out  'pragma_gen_fastio',
out  'pragma_gen_deepcopy',
out  'pragma_gen_all_off',
out  'pragma_c_only']
out
```

These two objects contain the entirety of Pex's internal state during compilation, though the only attributes you'll probably ever touch are: `env.cc`, `env.link`, and `scope.pragma_*`.

For example, to add a `-Wall` flag for the C compiler, do:

```
%whencompiling: env.cc.insert(1,'-Wall')
```

See the next section for how to use this C compilation configuration machinery to link with external C code.

14.1 Pragmas and the scope Object

The scope object contains, among other things, the various pragmas (see D for full listing) that you can use to control what happens during compilation. The scope object has slightly different semantics than env, any changes you make to it have effect in your current scope (e.g. current indentation level), and any child scope, but not outside. For example:

```
%whencompiling: scope.pragma_gen_strmeth = False

cdef class I:
        pass

cdef class II:
        pass
```

setting scope.pragma_gen_strmeth=False in the topmost scope, turns off generation of the __str__ method (see 9.2.2) for the entire module – neither class I or II will have it, whereas the following:

```
cdef class I:
        %whencompiling: scope.pragma_gen_strmeth = False
        pass

cdef class II:
        pass
```

turns it off for class I, but not class II.

14.2 pragma_c_only

Pex code can either generate fast C code, or slow code that calls the Python standard library, and it can be hard to tell just by looking at it. For example:

```
cdef int i
for 0<i<20:
    i+=3
```

is fast and maps directly to C code, whereas

```
for 0<i<20:
    i+=3
```

is slow, and runs at Python speeds because `i` is a python object. To ameliorate the situtation Pex allows you to set a pragma which will cause a compilation error if any code within the scope of the pragma generates calls to the Python library:

```
%whencompiling: scope.pragma_c_only=True
for 0<i<20:
    i+=3
```

```
PexPragmaCOnlyError
    A line of pex code that has been indicated as generating only fast C code
    using "%whencompiling: scope.pragma_c_only=1" appears to have generated
    calls to the Python library.

        pex line:

            1110_pragma_c_only_MUST_FAIL.px,2 | for 0<i<20:

        generated C:

              for (__pyx_3 = 0+1; __pyx_3 < 20; __pyx_3++) {
            __pyx_1 = PyInt_FromLong(__pyx_3); if (unlikely(!__pyx_1)) {__pyx_filename = __pyx_f[0]; __pyx_lineno
            if (PyObject_SetAttr(__pyx_m, __pyx_n_i, __pyx_1) < 0) {__pyx_filename = __pyx_f[0]; __pyx_lineno = 8
            Py_DECREF(__pyx_1); __pyx_1 = 0;



        suspect call:

            PyInt_FromLong

```

:-(`pragma_c_only` allows the following calls: `Py_INCREF`, `Py_DECREX`, `Py_xDECREF`, and `PyErr_Occurred`. The refcount calls are in some sense unavoidable whenever dealing with Python objects, the `PyErr_Occurred` is called after any function call that may have generated a python exception. You may take away your functions' ability to throw exceptions and get rid of this check whenever they are called (see 20), though being able to throw exceptions is good, and the checks are cheap.

15 Linking with External C Code

Here is an example showing how to use the functionality described in the previous section to setup your compilation to link with external C code.

```
main.sh
        #!/bin/sh
        set -x                                                  # echo shell commands
        (cd C_src && gcc -c C_module.c -o C_module.o)   # compile our C module
        $PEX prog.px                                           # run the pex module

C_src/C_module.c
        #include <stdlib.h>
        #include "include/C_module.h"

        vec *vec_create(unsigned object_size, unsigned length) {
                return malloc(object_size * length);
        }


C_src/include/C_module.h
        typedef void vec;
        vec *vec_create(unsigned object_size, unsigned length);


prog.px
        %whencompiling:
                env.cc.append('-I./C_src/include')
                env.link.append('C_src/C_module.o')

        cdef extern from "C_module.h":
                ctypedef void vec
                vec *vec_create(unsigned object_size, unsigned length)

        cdef vec *v
        v=vec_create(4,100)
        print '0x%x'%<int>v
        print
```

```
out + cd C_src
out + gcc -c C_module.c -o C_module.o
out + /res/home/dg/git/plat/pex/pex -F prog.px
out
out 0x943e0d8
```

Refer to Pyrex documentation for more complete information about interfacing with C code:

16 Backtraces on SIGSEGV, SIGFPE, SIGBUS, and SIGABRT

Pex attempts to make the vigorous experience of a C coredump a tad more civilized. You, gentle reader, will of course never encounter such a bug in your own code, but in case you ever have to run someone else's, Pex registers a signal handler for SIGSEGV, SIGFPE, SIGBUS, and SIGABRT that attempts to recover the C stack and print the backtrace:

```
1 cdef POOF():
2         cdef int *null_pointer=NULL
3         print null_pointer[0]
4
5 def func(): POOF()
6
7 def main(): func()
```

```
out SIGSEGV has occured. DON'T PANIC!!!!!
out Put on your C hat, about to attempt a backtrace. The memory is corrupted,
out so anything may happen next, for example this backtrace attempt may hang...
out ---- BEG BACKTRACE -----------
out    Containing Executable File       Instruction Addr             Closest Symbol
out
out    /res/home/dg/git/plat/pex/doc/examples/1400_coredump_MUST_FAIL.so 0x6A17     __pyx_pf_231400_coredump_MUST_FA
out
out    /opt/lib/libpython2.6.so.1.0     0x67DF7                    PyCFunction_Call
out    /opt/lib/libpython2.6.so.1.0     0x26F9C                      PyObject_Call
out    /opt/lib/libpython2.6.so.1.0     0xBE9C4       PyEval_CallObjectWithKeywords
out    /opt/lib/libpython2.6.so.1.0     0x2704C                 PyObject_CallObject
out
out    /res/home/dg/git/plat/pex/doc/examples/1400_coredump_MUST_FAIL.so 0x6AB8     __pyx_pf_231400_coredump_MUST_FA
out
out    /opt/lib/libpython2.6.so.1.0     0xC5E6D                  PyEval_EvalFrameEx
out    /opt/lib/libpython2.6.so.1.0     0xC7114                   PyEval_EvalCodeEx
out    /opt/lib/libpython2.6.so.1.0     0xC7193                     PyEval_EvalCode
out    /opt/lib/libpython2.6.so.1.0     0xEAF3A                   PyRun_FileExFlags
out    /opt/lib/libpython2.6.so.1.0     0xEB274             PyRun_SimpleFileExFlags
out    /opt/lib/libpython2.6.so.1.0     0xEB56A                PyRun_AnyFileExFlags
out    /opt/lib/libpython2.6.so.1.0     0xF760D                             Py_Main
out
out    python                           0x592                                 main
out
out    /lib/libc.so.6                   0x15E9C                  __libc_start_main
out
out    python                           0x4A1                              (null)
out                                                                       [START]
out ---- END BACKTRACE -----------
out
out If you want to examine the source code for these functions, the corresponding
out executable file (either *.so or the program itself) must have been compiled
```

```
out with '-g'. If it wasn't, you can do so now; you don't have to rerun the program,
out if the source remained the same, the address for the function will be valid.
out
out For example, to get at
out
out     ./mod.so                          0x247A                    __pyx_pf_3mod_func
out
out with addr2line, from the shell do
out
out         $ addr2line -e ./mod.so 0x247A
out
out with gdb,
out
out         $ gdb ./mod.so
out         (gbd) l *0x247A
out
out NOTE: The last stack frame may be a little off.
out
out Kaplah! Heghlu'meH QaQ jajvam.
out
out sh: line 1: 23677 Segmentation fault      (core dumped) /res/home/dg/git/plat/pex/pex -F 1400_coredump_MUST_FAIL
```

Gotchas

:-(At the time this signal handler is called, memory is corrupted, and so anything may happen. For example, as has been observed a few times, "attempting backtrace.." may get printed, and then the program will hang. If you prefer the normal core dump behavior, turn this functionality off by adding `-nobt` to the Pex command line.

:-(This functionality is on by default, you may turn it off by using `-nobt` flag on the command line (see 17 about command line usage), but if you import any Pex modules not so compiled, the signal handler will get registered again.

17 Command Line Usage

```
pex [options] [FILE{.px|.so}]

  v1.2

  -h(elp)       print this message
  -clean        remove all generated files in the current directory  (.pyx .pxd .c .o .so .pxi .dep .pyx.html .c
  -build        build all .px files in the current directory
  -list         list .px files and all their existing build products
  -R            recurse down directories, applies to -build, -clean, -list, and -covreport
```

```
-F            force build
-v[#|all]     trace level
-d(ebug)      takes out '-O6' flag and adds '-g' to the gcc command line
-g            adds '-g' to the gcc command line
-b            compile with ndarray bounds checks
-c            compile for code coverage (also can set PEXCOVCOMPILE environment variable)
-covreport    generate .px.cov files after a compiled for code coverage run
-u(nsafe)     remove '-fno-strict-aliasing' from gcc command line potentially
              speeds things up, but also, potentially gives incorrect behavior
-nobt         turn off backtrace printing on SIGSEGV, SIGFPE, SIGBUS, and SIGABRT
-W            print warnings
-a            annotate, produce FILE.pyx.html showing how each line is converted into C
```

To see what Pex is doing, in exhausting detail, run it with full verbosity:

```
$ pex -vall main.px
```

If Pex is not run directly, but is involved in a `pex.pimport()` call from python, you can still turn on verbosity by setting an environment variable:

```
#!/usr/bin/python

import pex
mod = pex.pimport('mod')
```

```
$ export PEXVERBALL=1
$ python main.py
<lots of output follows>
```

18 Compiling for Code Coverage

You may also compile such that after running you get code coverage information - the number of times each line of source code was executed. For example, suppose you have the following pex file `main.px`:

```
def main():
    for i in range(5):
        x=i
```

You compile it with the '-c' flag to turn on code coverage:

```
$ pex -c main.px
* pex.build  main.px +code_coverage
* pex.build  done
```

The command above also runs the file, which produces various statistic files. Once the run is done, run the following command to get the report:

```
$ pex -covreport main.px
* coverage  : 100% (   3/   3)   <CWD>/main.px.cov
```

The display shows that 3 out 3 source lines in main.px were executed. If you look in `main.px.cov`, you will see the counts for each line:

```
$ cat main.px.cov
         1:    1:def main():
         6:    2:    for i in range(5):
         5:    3:        x=i
```

If the count is `#####`, the line was never executed. If the count is `-`, the line is not executable (for example, a comment). You may also turn on compiling for code coverage using the environment variable `PEXCOVCOMPILE`:

```
$ pex -clean main.px
$ PEXCOVCOMPILE=1 python -c "import pex; pex.pimport('main')"
* pex.build  main.px +code_coverage
* pex.build  done
```

The `-covreport` flag has the same semantics as `-build` and `-clean`, with no arguments it runs on every pex file in the current directory, with `-R`, it will recurse.

19 Distributing Code

Recall that from a single `main.px`, Pex compiles `main.c`, `main.o`, `main.dep`, `main.pyx`, `main.pxd`, and `main.so`. All you need to distribute to run your code are the `.so` files, you can say the following to run main:

```
$ pex main.so
```

If you have many modules, you still only need to distribute their `.so` files, but you must maintain the same directory structure, you couldnt put all the `.so` files in the same directory, and you must continue to have `__init__.py` files in your package directories.

If you wish to have new pex modules compiled against the ones you are distributing, that is new pex modules `pimport`ing the ones you are distributing, `.so` files are not sufficient, you must also distribute the `.pxd` files.

For example, you start out with the following files:

```
main.px  pkg/__init__.py  pkg/mod.px
```

when you compile with pex, you get:

```
main.c    main.px   main.so           pkg/mod.c    pkg/mod.px
main.dep  main.pxd  pkg/__init__.py   pkg/mod.dep  pkg/mod.pxd
main.o    main.pyx  pkg/mod.o         pkg/mod.pyx  pkg/mod.so
```

If you distribute:

```
main.so  pkg/__init__.py  pkg/mod.so
```

you may run `main.so` using Pex, and you may import `main` from python modules. If you distribute:

```
main.pxd  main.so  pkg/__init__.py  pkg/mod.pxd  pkg/mod.so
```

you can then also `pimport` main from other pex modules.

20 Exceptions

Pyrex handles exceptions by checking a global error variable after every call to a `def` or a `cdef` function. In case this is a performance problem, or a function can not possibly throw an exception, Pex allows you to turn this check off by adding `except exits` to the function prototype:

```
1 cdef void func() except exits:
2     assert 0
3
4 try:    func()
5 except: print "caught exception"
```

```
out PEX: Unpropagated exception has occurred (because its inside a cdef func that
out      opted to die on an exception, using 'except exits:')
out   File"main.pxd", line 4, in main.func
out AssertionError
```

The `except exits` clause to `func` does two things, first there is no error variable check after calls to `func`. Second, any exceptions that occur inside `func`, or any functions it calls, cause the program to exit with an error message, as can be seen in the above example. Note that

this is different from Pyrex, where an unpropagated exception prints an error message, but the program continues to run.

The `except exits` clause also makes sense for functions that can not possibly throw a python exception, like those in external C libraries (see E for examples of this use in `builtin.pxi`). See Pyrex documentation about error return values for a more complete discussion of the `except` clause in function prototypes.

20.1 Using Exceptions in Fast, Low Level Code

One can use exceptions, with all of their niceties, even in low level C code, without any performance penalties. Generating and throwing exceptions is expensive, but since it only needs to happen on an error condition, which can be checked for with an efficient if statement, this does not slow down the common case:

```
cdef void low_level_io():
    cdef FILE* f

    f = c_fdopen(-1, "r")
    if f == NULL:
        PyErr_SetFromErrno(PyExc_IOError); return

def main():
    low_level_io()
```

```
out Traceback (most recent call last):
out   File "/res/home/dg/git/plat/pex/pex", line 334, in <module>
out       332 |
out       333 | tr(1,"calling %s.main()"%name)
out     >> 334 | exit_code=module.main()
out   File "1602_exc_for_low_level_code_MUST_FAIL.px", line 9, in 1602_exc_for_low_level_code_MUST_FAIL.main
out         7 |
out         8 | def main():
out     >>  9 |     low_level_io()
out   File "1602_exc_for_low_level_code_MUST_FAIL.px", line 6, in 1602_exc_for_low_level_code_MUST_FAIL.low_level_io
out         4 | f = c_fdopen(-1, "r")
out         5 | if f == NULL:
out     >>  6 |     PyErr_SetFromErrno(PyExc_IOError); return
out <type 'exceptions.IOError'>: [Errno 9] Bad file descriptor
```

Gotchas

:-(If the return type of a function is a python object, `except exits` has no effect. In the following example, the return type is omitted and thus defaults to a python object. As a result, the exception is plumbed through:

```
cdef func() except exits:
    assert 0

try:    func()
except: print "caught exception"
```

```
out
out caught exception
```

20.2 Exception Traceback Formatting

Suppose you have the following two files:

```
mod.px
    def func():
        assert 0

main.px
    %pimport mod

    def main():
        mod.func()
```

Running `main.px` with python's standard traceback formatting gives you the following output:

```
% pex main.px
Traceback (most recent call last):
  File "main.pyx", line 92, in main.main
    mod.func()                      ## main.px,6
  File "mod.pyx", line 88, in mod.func
    assert 0                            ## mod.px,2
AssertionError
```

The line numbers refer to `.pyx` files, and the traceback can be significantly harder to read in more complicated situations than the one given here. For this reason pex registers a custom `sys.excepthook` that gives a nicer output:

```
Traceback (most recent call last):
  File "main.px", line 4, in main.main
    mod.func()
  File "mod.px", line 2, in mod.func
    assert 0
AssertionError
```

If you want to access this nicer formatting when custom printintg tracebacks, and `.px` files are involved, you may use the two functions `pex.print_exc([file=sys.stderr])` and `pex.format_exc()`. They are analogous to python's `traceback.print_exc()` and `traceback.format_exc()`, except they do not support the `limit` argument.

21 Efficiency

`cdef bool x = True` is slow, use `cdef bool x = cTrue`. Also note that `x is True` is slow, as `x` gets upcast to python. SLOW

Be careful with loops, `for 0<=i<n` is slow if `i` is not a `cdef`'d int. SLOW

If you have an expression that is all `cdef`'d variables, it will run at C speed; however if you happen to have a single variable in the expression that is a python object, things will automatically and silently be upcast to Python and will become slow. It is easy for this to happen, and hard to detect, since by just looking at the expression you can not tell what its constituent variables are – `cdef` variables or python objects. See B.2.2 for more information on this topic. Your best bet, if things are not running as fast as you expected, is to look at the generated C code, and see what is actually happening. SLOW

Use `except exits` when declaring prototypes for external C functions, see 20 for full discussion. SLOW

Be careful with object attribute access, `x.a` will run two orders of magnitude slower, if `x` is a python object, rather than a `cdef` class. See B.2.3 for more information. SLOW

Make sure to give a `cdef` function a return type. If you don't specify one, it defaults to a python object, and will return the Python `None`, even if you don't have any returns. SLOW

21.1 Annotate Mode

Most of the inefficiencies creep in when you use code that triggers the use of the python runtime. This can be hard to detect just by looking at the code, for example you may be using a variable that you forgot to cdef as an int, and which therefore is a python object. Pex can generate an html file for your `.px` file that shows how your code is expanded into C code:

```
$ pex -a module.px
$ firefox module.pyx.html
```

The html file shows your code lines, each colored in a different shade of yellow, the more yellow, the more C lines your pex code line was expanded into, the slower it probably runs. You may double click on any line to see it's actual expansion into C code.

:-(Due to a buglet in this feature, you may not use an empty except clause if you wish to generate the html file. Instead of this,

```
except:
```

do this,

```
except Exception:
```

22 Gotchas

:-(Accessing uninitialized cdef variables: the following mistake virtually guarantees memory corruption, and is so easy to make, that even after reading this scary warning, you will make it multiple times, but at least this author's consciousness will be clear. The following snippet causes a coredump: DEATH!

```
1 cdef class item:
2     cdef int a
3
4 cdef item x
5
6 x.a
```

What happened, is that `x` was accessed before it was instantiated, we should have done an `x=item()` first. Since we did not, the value of `x` was `None`, which we then proceeded to access as if it had type `item`. `None` was cast to type `item`, and then the memory address of where attribute `a` would have been was accessed. This got us a random memory location; had we written to the attribute, we would have corrupted the `None` object, and had we called a cdef'd method of `item`, we would have called a random memory location. BE SURE TO INSTANTIATE ALL YOUR CDEF OBJECTS, INCLUDING NDARRAYS, BEFORE ACCESSING THEM.

:-(The semantics of the Python `__path__` are not supported.

:-(The `pex` directory must be installed in your Python path in order for the compiled Pex modules to be importable and usable.

:-(Pex does not handle Mac '\r' terminated files.

:-(The mod operator % gives different results for negative numbers depending on whether the variable you apply it to is a C variable or a Python variable. Therefore if your negative number is in a C int you'll get one answer, if it is in a Python int you'll get another:

```
cdef int x = -10
y          = -10

print '-10 % 3 ==', x % 3, 'for C int'
print '-10 % 3 ==', y % 3, 'for Python int'
```

```
out
out -10 % 3 == -1 for C int
out -10 % 3 == 2 for Python int
```

This can be especially confusing, because at times it's hard to keep track of whether the result of an expression is a C value, or a Python value (see B.2.2 for a description of how and when Pyrex converts between C and Python variables).

:-(In some situations, Pyrex does not complain if you define the same variable with two different types, the following does not cause a compile or runtime error:

```
cdef char x
cdef int  x
```

In the above, the last declaration wins.

:-(Pyrex does not directly warn if the name of a class attribute clashes with a member function, but instead gives you mysterious errors:

```
cdef class item:
    cdef int value
    def value(self): pass
x = item()
x.value()
```

```
out PyrexError
out     <snip>
out     main.pyx: Cannot take address of Python variable
```

:-(Static variables are not supported, use global variables instead. Global variables have the same semantics as C globals.

:-(Do this:

```
cdef class item:
    pass
```

not this:

```
cdef class item: pass
```

or you'll get a mysterious `PexUnexpectedInternalError`.

:-(If you turn off generation of all special class functions (see 9), and have nothing in the class except variable declarations, the variable declarations will get sent to the header file, the class body will become empty, and you'll get a confusing syntax error. Do this:

```
%whencompiling: scope.pragma_gen_all_off = True
cdef class item:
    cdef int x
    pass
```

instead of this:

```
%whencompiling: scope.pragma_gen_all_off = True
cdef class item:
    cdef int x
```

:-(To assign a character literal, you must use the following hack:

```
cdef char c = 'a'[0]
```

Though contorted, the code still runs at C speed.

:-(Be careful with multi-line statements, they can be fragile due to Pex's lack of parsing. For example, the following causes a syntax error:

```
def func(a, # informative comment
         b):
```

because behind the scenes the line gets converted into `def func(a,# informative comment b):`.

:-(Be careful with ':' and ';', don't try to combine too many things into one line, for example the following is a syntax error:

```
except: do_something(); raise
```

:-(Be careful with ':', '{', and '}'. Their liberal use, while legal Python and Pyrex syntax, may easily screw up Pex's regular expression based processing, for example the following wouldn't be recognized as a function prototype:

```
cdef func(a={'key': 'val'}):
```

:-(If you compile your Pex modules optimized, you want to also pass the `-fno-strict-aliasing` to the compiler – this is the default. It slows things down slightly, but without it, with a non-trivial probability, Pyrex's object system breaks (see 14 about how to configure the compilation).

:-(If you get a "⟨some_type⟩ is not the right type of object" error, clean and rebuild everything, this may mean that for some reason Pex's rebuild mechanism has failed.

:-(Triple quoted strings, such as """, are supported, but are munged. They are meant to be used as block comments, if you print them out, they'll be different from what you wrote (new lines and indentantion striped out, and {} replaced with ()).

:-(This is a big one, if you have a module with a cdef class, and this module also has a function that takes this cdef class as a typed argument, you can not import this module through a package – you can do `import mod`, but not `import pkg.mod`. The following breaks:

```
pkg/__init__.py

pkg/mod.px
        cdef class item:
                pass

        cdef func(item x):
                pass

main.px
        %pimport pkg.mod
```

```
out Traceback (most recent call last):
out     <snip>
out TypeError: C function pkg.mod.func has wrong signature <snip>
```

As a work around, you may pass in a generic Python object, and then unpack it into the proper type:

```
pkg/__init__.py
```

```
2
3 pkg/mod.px
4         cdef class item:
5                 pass
6
7         cdef func(obj):
8                 cdef item x = obj
9
10 main.px
11        %pimport pkg.mod
```

:-(In the following declaration:

```
1 cdef class item:
2     cdef int i
```

the attribute `i` is NOT static, it is a regular class attribute. It is easy to confuse it for a static, because in Python, static variables are declared as follows:

```
1 class item:
2     i=3
```

:-(If you have a cdef attribute in a derived class, with the same name as a cdef attribute in the base class, the base class attribute is not replaced, both still exist:

```
1  cdef class base:
2      def __init__(me):
3          cdef int me.x=3
4
5      def func(me):
6          print 'base.x',me.x
7
8      def set_base(me, int x):
9          me.x=x
10
11 cdef class derived(base):
12     cdef int x
13
14     def func(me):
15         base.func(me)
16         print 'derived.x',me.x
17
18 cdef derived d = derived()
19
20 d.func()
21 d.x = 7
```

```
22 print
23 d.func()
```

```
out
out base.x 3
out derived.x 0
out
out base.x 3
out derived.x 7
```

:-(Here is a confusing error,

```
1 cdef class item:
2     cdef meth(me, int *ip): pass
3
4 cdef int *ip
5
6 x = item()
7 x.meth(ip)
```

```
out PyrexError
out     python cython.py "main.pyx"
out
out     Error converting Pyrex file to C:
out     ------------------------------------------------------------
out     <snip>
out     cdef int *ip   ## main.px,4
out     x = item()   ## main.px,6
out     x.meth(ip)   ## main.px,7
out             ^
out     ------------------------------------------------------------
out     main.pyx:170:9: Cannot convert 'int *' to Python object
```

What happened here, is that `x` is untyped (see B.2.4 for full discussion of typed vs. untyped objects). As such, it is a regular python object, and does not expose the `cdef meth()` method. Thus Pyrex attempts to call it's `def meth()` method. The `def meth()` does not exist, and a call to it will cause an attribute error, but in order for this to happen, the call has to be attempted, and in order to be attempted, the arguments for the call have to be marshalled into Python objects, thus Pyrex attempts to convert the `int *ip` into a Python object, which causes the error we are seeing. To fix, declare `x` typed: `cdef item x`.

:-(Suppose you iterate over a generator: `for i in generator_func():` (see Python documentation about what a generator is, here it is a function that uses `yield`). There appears to be a

bug in Pyrex where exceptions thrown from inside a generator are ignored, so in this case if `generator_func()` throws something, you'll never see it.

:-(You have to be careful about explicitly casting a python object to a C number. You may think you are getting the value of the object, if for example it is a python integer, but you'll really only be getting it's memory address.

```
x = 17
cdef int xint = <int> x
print x, xint
```

```
out
out 17 164584608
```

23 Crossplatform Status

Pex is primarily targeted for Unix, though everything works on MacOS, except for backtraces on SEGFAULT. Pex is probably not too far from working on Windows with gcc, but does not at the moment.

24 Acknowledgements

We'd like to thank Greg Ewing for creating Pyrex, and making Pex possible. Through our usage of Pyrex, we've grown continually more impressed and are now left with a firm conviction that it is a technical tour de force. We'd like to thank the folks at the Cython project for augmenting Pyrex with many convenient and useful features. Finally, we'd like to thank our colleagues at PNYLAB for gallantly blowing themselves up on a wide variety of exciting mines as Pex was being developed, and who continue to deal stoically with some of Pex's more rude quirks.

25 Conclusion

This version of Pex is our first attempt at creating a language that achieves performance of C, but retains most of the elegance and clarity of Python. As a first attempt it is rough around the edges, but we believe it is immediately useful, and are shifting our entire codebase into Pex. Pex gives the ability to write C fast numerical loops without any special contortions, and C fast large scale projects through the virtue of having objects with C fast attribute and method access. Through Pex's mirroring of the Python import mechanism, large projects do

not need to have any build infrastructure, and much of the boiler plate code generally required for large projects (e.g. to quickly serialize objects to disk or over sockets), is automatically generated. This, largely performance oriented functionality, lives alongside the ability to mix in Python code and classes, and thus get the full niceties of the Python language. In fact, things mix together so seamlessly, that it can be a problem at times to figure out whether you are writing C fast code or not.

It is our belief, reinforced throughout our time working on Pex, that this is something missing in the world, and not for any good technical reasons. It is eminently possible to have a language that gets down to the iron, runs at C speeds, and has no surprises in generated assembly, but at the same time guides you along to a clear, succinct and correct expression of complicated systems and algorithms. We think Pex is a good first step towards addressing this lack.

APPENDIX

A Wrapping `cdef` functions

The current version of Pex does not allow for naked cdef functions to be directly visible in the Python environment. This limitation is a historical accident: Pex was initially layered over an early version of Pyrex that did not have this feature. Indeed, exporting naked functions from one Pex module to another is only partially supported.

For all our examples, we consider how to wrap the Pex/C routine `sqrt`.

```
1 cdef extern from "math.h":
2         double sqrt(double x)
```

A.1 Wrapping `cdef` functions: the slow and simple way

The standard method to make cdef functions visible to the Python environment is by defining a Python interface to the function within the Pex file. For example, one may write:

```
1 def py_sqrt(x):
2     return sqrt(x)
```

One can then import `py_sqrt` into the Python world; calls to `py_sqrt` are thus forwarded to `sqrt`. The reason this works is that when a Pex file is compiled, its Python functions have access to all of the cdef functions within this file. Then, when the compiled module is imported into a Python environment, the Python functions (and methods) are made visible to the Python environment.

This indirection has a price: one has to go through a thin layer of Python code to access the `cdef` function. This "thin" layer may very well take longer than the actual invocation of the `cdef` function itself. However, this cost is often immaterial: if one is wallowing in Python, the cost of a single extra Python call is comparatively small.

However, there are circumstances in which one may sensibly wish to avoid any Python overhead. For example, given a function f, one may wish to compute $f(x)$ for each value x in some ndarray v, and store the result in another ndarray, *out*. Such a basic mapping operation is a clear candidate for implementing in Pex. One would like to make a mapper, pass it a function, and have it construct an object that maps vectors to vectors, without going through any Python layers (aside from the initial call).

A.2 Wrapping cdef functions via classes

To avoid the Python layer, we can wrap cdef functions as instances of class objects. Thus, we can wrap sqrt as follows:

```
cdef class sqrt_class:
    def __init__(me): pass
    cdef double apply(me,double x): return sqrt(x)
    def __call__(me,x): return me.apply(x)

py_sqrt=sqrt_class()
```

We can then import py_sqrt and use it as we would any Python function (note that f(x) is syntactic sugar for f.__call__(x)).

The problem with this simple approach is that we cannot easily pass py_sqrt to some other general Pex routine, since each wrapped function has a distinct type. However, this problem is remedied using subclassing.

A.3 Obtaining generality via subclassing

Since Pex has a rigid type system, we have to be a little bit cleverer in order to obtain general routines. We mentioned that it would be nice to wrap a routine so that it would operate on vectors at a time (a map operation). Due to typing issues, we cannot hope to make a truly general mapping object. But we *can* be reasonably general. Suppose we want to take a function f that maps a double to a double, and create a new (mapping) function fv that maps an ndarray of doubles to an ndarray of doubles. We first define a dummy class, function_dd, that captures the idea of mapping a double to a double:

```
cdef class function_dd:
    def __init__(me): pass

    cdef double apply(me,double x): return 0.0 #dummy routine

    def __call__(me,x): return me.apply(x)
```

Then, we can wrap the sqrt function with its own class, but specify that this is a subclass of function_dd:

```
cdef class sqrt_class(function_dd):
    cdef double apply(me,double x): return sqrt(x)

py_sqrt=sqrt_class()
```

We can use `py_sqrt` as before, and it will compute the specific function we want. However, since it is a subclass of `function_dd`, other routines can treat it as a generic function from doubles to doubles. Thus, we can write:

```
cdef class mapper:
    def __init__(me,function_dd f):
        cdef function_dd me.f
        me.f=f
        return

    cdef void vapply(me,
                     ndarray<double 1d> v,
                     ndarray<double 1d> out):
        cdef int i,n
        cdef function_dd f
        f=me.f

        n=v.dimensions[0]

        for i from 0<=i<n:
            out{i}=f.apply(v{i})

        return


    def __call__(me,v,out=None):
        if out is None: out=numpy.zeros_like(v) #note: must import numpy
        me.vapply(v,out)
        return  out
```

After importing `py_sqrt` and `mapper`, we can set `vsqrt=mapper(py_sqrt)`, giving a routine that computes the square root of each element of an ndarray (of doubles).

It should be noted that we still have an intermediate layer - the wrapped function must call its apply method which then calls the actual function. However, this layer is much, much faster than going through python.

A.4 Do we need all of these classes?

It seems wasteful to make an entire class object for each function we would like to wrap. We can be more economical by making a single type, and performing invocations of this type within the pex file.

```
#define a raw function (not an object!) from double-to-double
ctypedef double (*raw_dd)(double)

```

```
cdef class wrapper(function_dd):
    def __init__(me):
        cdef  raw_dd me.f
        return

    cdef set(me,raw_dd f):
        me.f=f
        return

    cdef double apply(me,double x):
        return me.f(x)

cdef wrapper px_sqrt
px_sqrt=wrapper() #creates a wrapper object
px_sqrt.set(sqrt) #sets it to the specific function
py_sqrt=px_sqrt #py_sqrt is a python object
```

Most of this example is straightforward. We create a specific instance of the wrapper object, and set it to a specific function. But why are we using two variables, `px_sqrt` and `py_sqrt`? We need to explicitly declare a variable to be of type `wrapper` so we can access the set method. Then we need to make it visible to the python world, so we have a second variable, `py_sqrt` that is a python object (this is the default type when a variable has not been explicitly typed).

Now, we should be able to have a single expression that generates the wrapper object as a subexpression, then assigns it without ever being assigned through an extra variable. However, in our attempts, strange things happened: The compiler perversely tried to convert an object of type `raw_dd` to a python object, which it cannot do. Using explicitly typed intermediate variables seems to guide the compiler past this pathology.

With such ready access to the raw functions, we can rewrite the `mapper` object to be much more efficient, achieving something much closer to full speed.

```
cdef class mapper2:
    def __init__(me,wrapper function):
        cdef raw_dd me.f
        me.f=function.f
        return

    cdef void vapply(me,
                     ndarray<double 1d> v,
                     ndarray<double 1d> out):
        cdef int i,n
        cdef raw_dd f
        f=me.f

        n=v.dimensions[0]
```

```
15
16          for i from 0<=i<n:
17              out{i}=f(v{i})
18
19          return
20
21      def __call__(me,v,out=None):
22          if out is None: out=numpy.zeros_like(v)
23          me.vapply(v,out)
24          return  out
```

:-(While this method is certainly more efficient, the resulting objects will not be pickleable. Indeed, to compile this code, we needed to set the following pragmas at the beginning of the file:

```
1 %whencompiling: scope.pragma_gen_fastio=False
2 %whencompiling: scope.pragma_gen_dictcoercion=False
3 %whencompiling: scope.pragma_gen_pickle=False
```

Thus, any object that is built up of such objects will be unpicklable. **Use this method at your peril!**

A.5 Wrapping Python objects as Pex objects

The `mapper` function we have created is very particular in the type of the function it accepts. But suppose we have a Python object that maps doubles to doubles. Why can't we use the `mapper` object on it? Let us ignore the obscenity of performing ones trivial bookkeeping at C speed while computing the actual function at Python speed. We might argue that we shouldn't have to to the trouble of reimplementing `mapper` in Python, or that we are quickly prototyping in Python a function that we later intend to implement in Pex.

The problem is that the `mapper` object wants an object of a particular type (`function_dd`), and even if the Python object behaves like a `function_dd` object should, it just doesn't smell right. However, we can make a simple wrapper that converts an arbitrary object to a `function_dd` object:

```
1 cdef class python_wrapper(function_dd):
2     def __init__(me,obj):
3         cdef object me.obj
4         me.obj=obj
5         return
6
7     cdef double apply(me,double x):
8         return me.obj(x)
```

Note that whenever `me.obj(x)` is evaluated, a run-time type check of the answer is performed to make sure that it is a double.

B What is Pyrex?

This section is not intended to be a standalone description of Pyrex, for that please refer to the official Pyrex documentation. Also keep in mind, that as described in the abstract, Pex sits on top of Cython, a fork of the Pyrex project.

B.1 Simple

Pyrex, at it's simplest, does three things: it compiles your Python code to C, it allows you to mix C code with your Python code, and finally allows you to write classes – "cdef classes" – whose methods and attributes are accessed at C speeds, 1-2 orders of magnitude faster than methods and attributes of Python classes.

B.2 Elaborate

B.2.1 Python to C Compiler

Pyrex compiles your Python code to a sequence of C calls to the Python standard library, for example, Python code:

```
v=[]
v.append(3)
```

becomes something like the C code:

```
PyObject *v;
v = PyList_New(0);
PyList_Append(v, PyInt_FromLong(3));
```

Pyrex essentially reproduces the actions the Python interpreter would take at runtime, and writes them to a C file. This removes the overhead of the Python interpreter, however that by itself is usually not significant. However, Pyrex does other optimizations (many of them come from the Cython project), like making [] accesses go faster for list and tuples than for generic Python objects. All told, PyBench – the standard Python benchmark – runs 30% faster when compiled with Pyrex, than when run with Python. It is very probable that you can compile your Python code with Pyrex, with no changes, and have it run faster. Note, most Python code is valid Pyrex code, but not all (one example: `import *` is not supported).

You will likely have to make some small changes, see Pyrex Limitations, and also differences between Pyrex and Cython, because Cython removes some of these limitations.

B.2.2 Mixing in C Code

If you write the following:

```
x=3
y=4
x+y
```

both `x` and `y` are Python objects, thus `x+y` is an addition of two Python objects, and happens at Python speed, much slower than C speed. Pyrex allows you to declare and use C variables, much like you would in C, except you have to precede the declaration with "cdef":

```
cdef int x,y
x=3
y=4
x+y
```

Here, `x` and `y` are C ints, and the addition is an addition of C ints, passed by Pyrex through to the C compiler and running at C speed (essentially happening in one instruction), easily 2 orders of magnitude faster than the pure Python version given above.

In a situation when you have an expression where Python objects and C variables are mixed, Pyrex will first convert the C variables to Python objects, for example, here:

```
cdef int x=3
y=4
x+y
```

`x` is converted to a Python object, and `x+y` becomes a sum of two Python objects (notice that this expression now runs at Python speed). Pyrex is able to do this automatic conversion for most of the standard C types (see Pyrex documentation), including all of the ones supported by Pex (see 12).

Similarly, for functions, if you write the following:

```
def func(x):
        return x+1
func(3)
```

the call to `func()` incurs a lot of overhead because it goes through all of the Python function invocation machinery, the C call stack for such an invocation is something like:

```
PyCFunction_Call
PyObject_Call
PyEval_CallObjectWithKeywords
PyObject_CallObject
```

Like for variables, Pyrex allows you to prefix functions with "cdef" and write:

```
cdef func(x):
        return x+1
func(3)
```

This `func()` is a regular C function call, with no Python linkage overhead. Note: if you don't specify a return type, or an argument's type, a Python object is assumed; the above is equivalent to:

```
cdef object func(object x):
        return x
```

We can also specify the usual C types, e.g.:

```
cdef uint64 func(int x):
        return x+1
```

Here is an amazing thing about these `cdef` functions, you can take the following:

```
cdef void func(FILE *ioptr):
        cdef int x
        fread(&x, sizeof(int), 1, ioptr)
```

change it to:

```
cdef void func(FILE *ioptr):
        cdef int x
        if not fread(&x, sizeof(int), 1, ioptr): raise IOError
```

and have that Python `IOError` exception correctly plumb through the entire call stack, for essentially no overhead (a boolean check for every function invocation). If you've been around C, you are sure to understand this author's enthusiasm for this functionality.

Pyrex also supports `cdef`'ing structs, enums, externs, etc. You may use these in Pex, but not as `cdef class` attributes.

B.2.3 Differences between `class` and `cdef class`

Suppose you define a Python `class`:

```
class cl:
        def __init__(me): me.x=3
        def func(me): return 7
```

and do:

```
obj = cl()
obj.x
obj.func()
```

Here is what happened behind the scenes for `obj.x` and `obj.func()`, every Python class has a hash containing its methods and attributes, indexed by name. To execute `obj.x`, Python did a hash lookup for the string "x", and similarly for `obj.func()`, a hash lookup for the string "func". Thus, for any method or attribute access of a Python class, you incur the overhead of a hash lookup. Python also has another, faster, method for attribute access, to use which you specify a list of the attributes an a special `__slots__` attribute, however it is still far from C speed.

Pyrex allows you to avoid this overhead by using a "cdef" class (refered to in the Pyrex manual as "extension type"), here is the above example re-written to use it:

```
cdef class cl:
        def __init__(me): cdef int me.x=3
        cdef func(me): return 7
cdef cl obj=cl()
obj.x
obj.func()
```

Though things look similar, the `obj.x` and `obj.func()` are now accessed at the same speed as a C struct attribute, 1-2 orders of magnitude faster than the pure Python.

Though these "cdef" classes are in many respects similar to Python classes, they differ in several important ways. Here is a table summarizing some of these differences:

`class`	`cdef class`	
y	y	can have `def` methods
n	y	can have `cdef` methods
n	y	can have `cdef` attributes
y	n	can add new attributes on the fly
y	y	`def` methods visible to Python
na	n	`cdef` methods visible to Python
y	n	attributes visible to Python
y	y	can use `cdef` variables inside methods
na	y	`cdef` attributes visible to other `cdef` classes and Pyrex code ★
na	y	`cdef` methods visible to other `cdef` classes and Pyrex code ★

★ - These are the two primary reasons for having `cdef` classes.

Essentially, a `cdef class` is a more static creature than a plain Python `class`, its attributes are set once and for all at compile time – you wouldn't be too far wrong if you thought of it as a C struct.

Pyrex implements inheritance for these `cdef` classes, allowing you to derive subclasses, and also, amazingly, implements type polymorphism without sacrificing much in performance (it uses the same "vtable" approach that is used by C++). Type polymorphism is something you immediately discover you need once you start a medium to large size project that uses OOP. For example, suppose you want all your objects to have a fast `report()` method, which is used when you pass them to a generic report maker function. If you are coming from the Python side, this does not seem very hard; you can lookup the object's `report()` method dynamically, at runtime. With a compiled language, like Pyrex, all such lookups must be resolved in a more static fashion – this is where a lot of the speed up comes from.

The usual solution, providing you have type polymorphism, is to derive your classes from a common base class that implements a stub method for `report()`, and then in the derived class, override this method with something useful. Here is how you would do this in Pyrex:

```
1 import time
2
3 cdef class baseclass:
4         cdef void report(me): raise NotImplementedError
5
6 cdef class derived(baseclass):
7         cdef void report(me): print "report() of the derived class"
8
```

```
cdef do_report(baseclass b):
        print "report at time",time.time()
        b.report()      # ****************************************

d = derived()
do_report(d)
```

```
out
out report at time 1299875377.06
out report() of the derived class
```

The reason d'etre for all this type polymorphism machinery, is that the `obj.report()` call in the `do_report()` function runs at C speed, no matter the type of the passed in object (though note that that type has to derive from `baseclass`).

B.2.4 Difference between typed and untyped objects

When cdef class objects are created typed, like so:

```
cdef class item:
    pass

cdef item x = item()
```

you have access to `x`'s cdef methods, cdef attributes, and def methods. When created untyped:

```
x = item()
```

you lose access to all the cdef internals, and `x` appears as an almost regular python object, with the exception that you can not add new attributes. Here is a table summarizing the differences:

	CREATED WITH `cdef item x = item()`	CREATED WITH `x = item()`
access `cdef` attributes	y	n
access `cdef` methods	y	n
access `def` methods	y	y
add new attributes	n	n

B.2.5 Overriding {def,cdef,cpdef} methods with {def,cdef,cpdef} methods

Pyrex allows you to override a `def` method with a `cdef` method with no complaints, however the results when you run may be surprising. The advice is to never do such things, but just in case here is a table enumerating the possibilites:

```
cdef class A:   cdef class B(A):       what is printed?
  ? f():          ? f():            x=B()       cdef A x=B()
    print 'A'       print 'B'       x.f()       x.f()
--------------  ---------------     --------    --------
def             def                 B           B
def             cdef                A           A
def             cpdef               B           B

cdef            def                 B           A
cdef            cdef                err         B
cdef            cpdef               <compilation error>

cpdef           def                 B           A
cpdef           cdef                <compilation error>
cpdef           cpdef               B           B
```

C Differences Between Pex and Pyrex

C.1 Defining cdef Class Attributes

In Pex, you may define class attributes in the class preamble like so:

```
cdef class item:
    cdef int    i
    cdef double x
```

This is similar to Pyrex, though in Pyrex you would put these declarations in a separate, `.pxd`, header file.

Pex also allows you to declare attributes directly in the constructor:

```
cdef class item:
    def __init__(self):
        cdef int    self.i
        cdef double self.x
```

When you do so, you may also initialize the class attributes on the declaration line:

```
cdef class item:
```

```
    def __init__(self):
        cdef int self.i = 3
```

:-(Note that decorated ndarray attributes may only be declared in the `__init__` function, not in the class preamble.

C.2 Unsupported Features

(i.) `cdef` blocks are not supported

(ii.) `cpdef` functions are not supported

(iii.) C structs, pointers, function pointers, and C arrays are all supported in Pyrex, and in general are plumbed through to Pyrex by Pex, however none of these will play well as class attributes in Pex, at best making your classes immodest (see 9). In general all of these are considered out of bounds as far as Pex is concerned.

This not a complete list, these are just features the authors of Pex knew about, and didn't support. There are likely other unsupported features.

D Pragmas

You set the pragmas by putting the following in your code:

```
%whencompiling: scope.<some_pragma> = [True | False]
```

See 14 for a complete discussion of the `%whencompiling` directive. Here is the list of all the pragmas:

PRAGMA	PURPOSE	DEFAULT	SEE SECTION
`pragma_ndarray_type_check`	On/off flag for typechecking of decorated ndarrays	True	8.2.6
`pragma_ndarray_bounds_checks`	On/off flag for bounds checking of {} accesses to decorated ndarrays	False	8.2.7
`pragma_gen_all_off`	On/off flag to control the generation of all convenience methods for modest and unspoiled classes	True	9.2.1
`pragma_gen_strmeth`	On/off flag to control the generation of the `__str__()` method	True	9.2.2
`pragma_gen_equalmeth`	On/off flag to control the generation of the `_equal_()` method	True	9.2.3
`pragma_gen_richcmpmeth`	On/off flag for the generation of the `__richcmp__()` method	True	9.2.4
`pragma_gen_dictcoercion`	On/off flag for the generation of the `_todict_()` and `_fromdict_()` dictionary coercion methods	True	9.2.5
`pragma_gen_pickle`	On/off flag for the generation of the `__reduce__()` and `__setstate__()` pickling methods	True	9.2.6
`pragma_gen_fastio`	On/off flag for the generation of the `_fastdump_()` and `_fastload_()` fastio methods	True	9.2.7
`pragma_gen_deepcopy`	On/off flag for the generation of the `_deepcopy_()` method	True	9.2.8
`pragma_gen_hashmeth`	On/off flag for the generation of the `_hash_()` method	True	9.2.9
`pragma_c_only`	Generate errors if any slow calls to Python are generated	False	14.2

E Builtins

Pex brings certain C functions into your regular environment, for example you can do the following:

```
c_fprintf(c_stderr,"Here is the message: %s %d pm\n","the crow flies at",12)
```

```
out Here is the message: the crow flies at 12 pm
```

Here is the `.pxi` file that brings them all in:

```
cdef extern from "stdio.h":
        ctypedef struct FILE:
                char* _IO_read_ptr
                char* _IO_read_end
                char* _IO_read_base
                char* _IO_write_base
                char* _IO_write_ptr
                char* _IO_write_end
                char* _IO_buf_base
                char* _IO_buf_end
                int   _fileno

        FILE *c_stdout "stdout"
        FILE *c_stderr "stderr"

        int c_fflush "fflush" (FILE *stream) except exits
        int c_fprintf "fprintf" (FILE *file, char *format, ...) except exits
        int c_printf "printf" (char *format, ...) except exits
        int c_sprintf "sprintf" (char *s, char *format, ...) except exits

        int c_fwrite "fwrite" (void *, int, int, FILE *) except exits
        int c_fread "fread" (void *, int, int, FILE *) except exits
        int c_fclose "fclose" (FILE *) # no "except exits"
                                       # screws up PyFile_FromFile
        FILE * c_fdopen "fdopen" (int fd, char *mode) except exits

        int c_fgetc "fgetc" (FILE *) except exits
        char * c_fgets "fgets" (char *, int, FILE *) except exits

        int c_feof "feof" (FILE *) except exits

        int c_sscanf "sscanf" (char*, char*, ... ) except exits

cdef extern from "unistd.h":
        int c_read "read" (int fd, void *buf, int count) except exits
        int c_write "write" (int fd, void *buf, int count) except exits

```

```
cdef extern from "stdlib.h":
        long c_strtol "strtol" (char *, char **, int) except exits
        void* c_malloc "malloc" (int) except exits
        void c_free "free" (void*) except exits
        int c_atoi "atoi" (char*) except exits
        void c_abort "abort" () except exits
        void c_exit "exit" (int) except exits

cdef extern from "string.h":
        char *  c_strcpy  "strcpy"  (char *, char *) except exits
        char *  c_strncpy "strncpy" (char *, char *, int n) except exits

        int     c_strcmp  "strcmp"  (char *, char *) except exits
        int     c_strncmp "strncmp" (char *, char *, int n) except exits

        int     c_strlen  "strlen"  (char *) except exits

        void *  c_memcpy  "memcpy"  (void*, void*, int) except exits

cdef extern from "math.h":
        double M_E
        double M_PI

        double c_abs "fabs" (double) except exits

        double c_loge "log" (double) except exits
        double c_log10 "log10" (double) except exits
        double c_log2 "log2" (double) except exits

        double c_sqrt "sqrt" (double) except exits

        double c_pow "pow" (double,double) except exits

        double c_exp "exp" (double) except exits

        double c_floor "floor" (double) except exits

        double c_ceil "ceil" (double) except exits
        double c_round "round" (double) except exits
        double c_erf "erf" (double) except exits
        double c_erfc "erfc" (double) except exits
        double c_fmod "fmod" (double,double) except exits

cdef extern from "bits/nan.h":
        double NAN


cdef extern from "pex_builtin.h":
        double c_max(double,double) except exits
```

```
87          double c_min(double,double) except exits
88
89          int c_sign(double) except exits
90
91          #unsigned long long c_rdtsc() except exits
92
93  ctypedef void* c_function_pointer
94
95  cdef extern from "Python.h":
96          ctypedef struct PyObject:
97                  int ob_refcnt
98
99          ctypedef struct PyTypeObject:
100                 # more fields in here, but we don't care about them
101                 int tp_basicsize
102                 void* (*tp_new)(PyTypeObject*,void*,void*)
103                 int (*tp_traverse)(PyObject*, int (*visitproc)(PyObject*, void*), void*)
104
105         ctypedef struct PyObject:
106                 int ob_refcnt
107                 PyTypeObject *ob_type
108
109         int PyString_AsStringAndSize(object obj, char **buffer, int *length)
110         object PyString_FromStringAndSize( char *v, int len)
111         char* PyString_AsString(object string)
112
113         long PyInt_AsLong(object io)
114         int PyInt_Check(object o)
115         object PyInt_FromLong(long ival)
116         void Py_INCREF(PyObject *o) except exits
117         void Py_DECREF(PyObject *o) except exits
118         int Py_REFCNT(PyObject *o) except exits
119         int Py_SIZE(PyObject *o) except exits
120
121         object PyFile_FromFile( FILE *fp, char *name, char *mode,
122                                 int (*close)(FILE*))
123
124         int PyList_Append( object list, object item)
125
126         PyObject* Py_None
127
128         object PyErr_SetFromErrno( PyObject *type)
129         PyObject* PyExc_IOError
130
131         char* PyModule_GetName( PyObject *module)
132
133         long PyObject_Hash(object ob)
134
135 ctypedef char bool
```

```
136
137 cdef enum:
138         cFalse,cTrue
139
140 ctypedef unsigned char        uchar
141
142 ctypedef unsigned short       ushort
143
144 ctypedef unsigned int         uint
145
146 ctypedef long long            int64
147 ctypedef unsigned long long   uint64
```