Extending Python with Pyrex

Poll

1. How many people here have been to more than two Python conferences?

Poll

1. How many people here have been to more than two Python conferences?
2. How many have been to two or more talks of the form:
   - “I’ve got a prototype of a Python implementation that is ten times as fast as CPython?

Poll

1. How many people here have been to more than two Python conferences?
2. How many have been to two or more talks of the form:
   - "I've got a prototype of a Python implementation that is ten times as fast as CPython? I hope to finish it soon."
3. How many have been to ten or more such talks?

Apologetic groveling

- Experiments are important.
- Doing an experiment is better than doing nothing.
- Most Python users complain about performance but do nothing at all.
- Many experimenters have other projects that are very successful.

- ***But*** Pyrex is exciting because it is here and it works!

High level overview

- Pyrex compiles a Python-like language to C.
- This gives two advantages over Python:
  - Easy access to C types
  - Closer to C performance than Python
- Gives two advantages over C:
  - Easy access to Python types
  - Close to Python ease and flexibility
- Created by Greg Ewing

My advice

- You will be a much more proficient Pyrex programmer if you learn both C and Python.
- …but you could probably get by with cargo cult techniques…

Pyrex compared to …

- CXX and Boost.Python
  - Pyrex has no explicit C++ support
  - But also doesn't depend on C++ syntax
- SWIG
  - Because Pyrex is is Python-specific, by default Pyrex APIs are very "Pythonic"
  - The bridge code runs at C speed, not Python speed
  - SWIG can only bridge, not program itself
- Python2C, Starkiller and other Python compilers
  - Not as mature as Pyrex

On the other hand

- Pyrex has a big flaw compared to most other wrappers:
  - There isn't yet a tool to convert C headers to Python.
  - You need to re-declare structs, typedefs, function definitions etc.
- And there's that C++ issue…

Simple Pyrex function

```
def hello_world():
  print “Hello world”
```

- As you can see: it uses a quirky syntax where indentation implies nesting.

Manual compilation

```
paul:/tmp pprescod$ pyrexc hello_world.pyc
paul:/tmp pprescod$ ls hello_world.*
hello_world.c   hello_world.pyc
paul:/tmp pprescod$ wc hello_world.pyc
       3       5      41 hello_world.pyc
paul:/tmp pprescod$ wc hello_world.c
     183     579    5791 hello_world.c
```

Using Distutils

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext
import glob, sys

setup(name = 'helloworld',
  ext_modules=[
    Extension("hello_world", ["hello_world.pyx"]),
    ],
  cmdclass = {'build_ext': build_ext}
)
```

Using Pyximport

```
paul:/tmp pprescod$ python
Python 2.3 (#1, Sep 13 2003, 00:49:11)
[GCC 3.3 20030304 (Apple Computer, Inc.
  build 1495)] on darwin
Type "help", "copyright", "credits" or
  "license" for more information.
>>> import pyximport; pyximport.enable()
>>> from hello_world import hello_world
>>> hello_world()
Hello world
```

Generated code

- Aside from dozens of lines of boilerplate…

```
static char (__pyx_k1[]) = "Hello world!";
    static PyObject
    *__pyx_f_11hello_world_hello_world(
          PyObject *__pyx_self,
          PyObject *__pyx_args,
          PyObject *__pyx_kwds) {
    static char *__pyx_argnames[] = {0};
    if (!PyArg_ParseTupleAndKeywords(__pyx_args,
                    __pyx_kwds, "", __pyx_argnames))
    return 0;
… (continued)
```

Here is the code to print

```
/* "/private/tmp/hello_world.pyx":2 */
 __pyx_1 = PyString_FromString(__pyx_k1);
 if (!__pyx_1) {__pyx_filename = __pyx_f[0];
 __pyx_lineno = 2; goto __pyx_L1;}

 if (__Pyx_PrintItem(__pyx_1) < 0) {__pyx_filename
 = __pyx_f[0]; __pyx_lineno = 2; goto __pyx_L1;}

 Py_DECREF(__pyx_1); __pyx_1 = 0;
 if (__Pyx_PrintNewline() < 0) {__pyx_filename =
 __pyx_f[0]; __pyx_lineno = 2; goto __pyx_L1;}
```

Things to note

- Pyrex handles
  - checking error return codes from C functions
  - type conversions
  - reference counting
- Bear in mind that this talk will focus on how Pyrex is *different* than Python…but usually it is *very similar!*

Adding static type checking

```
def hello_world(char *message):
    print message
```

- Generated code:

```
static char *__pyx_argnames[] = {"message",0};
PyArg_ParseTupleAndKeywords(__pyx_args,
__pyx_kwds, "s", __pyx_argnames,
&__pyx_v_message)
```

cdef functions

- The Pyrex programmer can also generate a function that has a C calling convention rather than Python:

```
cdef extern int strlen(char *c)

def hello_world(message):
  print message, get_len(message)

cdef int get_len(char *message):
  return strlen(message)
```

Generated code for get_len

```
static int __pyx_f_11hello_world_get_len(char
  (*__pyx_v_message)) {
  int __pyx_r;

  /* "/private/tmp/hello_world.pyx":7 */
  __pyx_r = strlen(__pyx_v_message);
  /* Meaningless boilerplate deleted */

  return __pyx_r;
}
```

Generated code calling get_len

```
__pyx_1 = PyString_AsString(
                          __pyx_v_message);
if (PyErr_Occurred()) {
  /* error handling stuff deleted */
}
__pyx_2 = PyInt_FromLong(
  __pyx_f_11hello_world_get_len(__pyx_1));
```

Notes about cdef functions

- In either “regular” or “cdef” Pyrex functions you can mix and match Python types and function calls. Very few restrictions.
- “cdef” functions cannot be called directly from Python.
- Regular Python functions are a pain to call directly from C (need to convert everything to PyObjects)
- Calling between regular functions is slower (standard Python performance problem)

Pyrex variable declarations

- Basically like C, but prefixed with the word “cdef”

```
cdef int x, y
cdef float z
cdef char *s
```

Python objects

- By default, variables are of type “Python object”, with all of the dynamicity that implies.
- They can also be explicitly typed as “object”.

```
object o
o.foo() + o ** o.bar()
```

Pyrex does runtime casting

```
cdef conversions(int x,
                 object y):
    x, y = y, x
    return x,y

print conversions(1, 2)
print conversions(1, “2”)
    # causes TypeError
```

Pointers

- Similar to C (including pointers to pointers etc.), except there is no "*" deref operator: use [0] instead.

```
cdef double_deref(int **y):
    return y[0][0]
cdef int x # int
x = 5
cdef int *xptr # int ptr
xptr = &x
print double_deref(&xptr)#int ptr ptr
```

Casts

```
cdef int x, y
x = 1000
cdef void *xptr
xptr = &x
y = <int>x

print y
print <int>(&y)
```

Importing types and functions

- Declare a header for Pyrex to include and re-declare the relevant symbols in it.

```
cdef extern from "stdio.h":
  ctypedef struct FILE
  FILE *fopen(char *filename,
  char *mode)
```

Structures

```
cdef struct mystruct:
    int a
    float b
```

- Pyrex is more regular than C: Use "." for fields of structs and pointers to structs

```
cdef mystruct *m
m.a = 5
```

Other complex types

```
cdef union u:
  char *str
  int *x
cdef enum colors:
  red = 1
  green = 2
  blue = 3
```

Partial structure redeclaration

- Don't have to re-declare all struct members: just the ones you care about.

```
cdef extern from "stdio.h":
    ctypedef struct FILE:
        int _blksize
```

Typedefs

- Typedef works basically as in C:
  - `ctypedef unsigned long ULong`
  - `ctypedef int *IntPtr`
  - `ctypedef int size_t`

NULL

- There is a reserved word “NULL”.
- It is not the same as 0 or None.
  - 0 is an integer.
  - None is an object type
  - NULL is for pointer types.

Memory management

- Python objects are reference counted and garbage collected a la Python
- C types use manual C memory management
- Pyrex programmers don't need to think about refcounts
- Except…if they call Python/C API functions that steal or lend references.

Exception handling

- Pyrex adds exception handling to C.
- Even “cdef” (C) functions can throw exceptions.
- If the return type of the function is object, this works just like Python:

```
cdef object divide(int x, int y):
    if y==0:
        raise ZeroDivisionError
    else:
        return x/y
print divide(2,0)
```

This won't work

- Remember to think about types!

```
cdef object divide(int x, int y):
    return x/y

print divide(2,0)
```

- Division by 0 is not an error for C types!

What about C return types

- But what about when the return type is a C type?

```
cdef int divide(int x, int y):
    if y==0:
        raise ZeroDivisionError
    else:
        return x/y
print divide(2,0)
-------------
```

Generates:

```
Exception exceptions.ZeroDivisionError in
  'divide.divide' ignored
0
```

Except clause

- Functions can have an “except” clause.
- They declare that a particular return value indicates that an exception has occurred.

- Note that returning the value does not trigger the exception: you raise the exception as per usual.

Exceptions from C functions

```
cdef int divide(int x, int y) except -1:
    if y==0:
        raise ZeroDivisionError
    elif y<0:
        raise TypeError, "This function
  only  divides positive numbers"
    else:
        return x/y

print divide(2,0)
```

This also won't work

```
cdef extern FILE *fopen(char *filename,
  char *mode) except NULL
```

- Standard C functions don't throw Python exceptions (they don't call PyErr_SetString)
- Except clauses are only useful for functions that know about Python.

Conditional exception codes

- Sometimes any number is a valid return code.
- An “except?” clause tells Pyrex to *check* whether an exception was thrown.

```
cdef int  divide(int x, int y) except? -1:
    if y==0:
        raise ZeroDivisionError
    else:
        return x/y
print divide(-1,1) # works
print divide(2,0) # raises exception
```

Very conditional error codes

- Pyrex can check whether an exception was thrown no matter what the return value. (very inefficient!)

```
cdef void check_divisible(int x, int y) except *:
    if y==0:
        raise ZeroDivisionError
    print "Divisible! %s/%s" % (x, y)

check_divisible(-1,1) # works
check_divisible(2,0) # raises exception
```

Integer for-loops

- In order to get around the famous performance problems with range() (name lookup etc.), Pyrex has an integer syntax:

```
for i from 0 <= i < n:
    ...
```

Extension types

- An extension type is just like a Python class except that:
  - it has a more compact representation (more compact even than __slots__ instances)
  - it can directly contain C types (which __slots__ instances cannot)
  - it is a first-class type in the type system

Defining extension types

```
cdef class Shrubbery:
  cdef int width, height

  def __init__(self, w, h):
     self.width = w
     self.height = h

  def describe(self):
     print "This shrubbery is", \
                  self.width, \
               "by", self.height, "cubits."
```

Using Shrubbery from Python

```
x = Shrubbery(1, 2)
x.describe()
print x.width
  # exception --
  # not accessible from Python
```

Public and readonly attributes

```
cdef class Shrubbery:
    cdef public int width, height
    cdef readonly float depth
```

- These attributes are accessible from Python:

```
x = Shrubbery(1, 2)
x.describe()
print x.width # works now
x.depth = 5 # throws exception
```

Properties in Pyrex

```
cdef class Spam:
  property cheese:
     "A doc string can go here."
     def __get__(self):
     # Called when the property is read.
          ...
     def __set__(self, value):
     # Called when the property is written.
          ...
     def __del__(self):
      # Called when the property is deleted.
```

Using extension types

- Extension types can be treated as simply Python objects, but they also exist in the Pyrex type system:

```
def widen_shrubbery(Shrubbery sh,
  extra_width):
    sh.width = sh.width + extra_width
```

Careful!

- Extension types have __special__ methods just as Python types do, but they are sometimes subtly different.
- Read the Pyrex docs for more information.

Pyrex lacks some features

- Function and class definitions cannot occur in function definitions
- “import *” is banished.
- No generators
- No globals() and locals() functions

Other missing features

- (to be corrected eventually)
  - Functions cannot even be nested in conditionals
  - In-place operators (“+= ”, “-= ”) are not allowed
  - No list comprehensions
  - No explicit support for Unicode
  - New division syntax

Non-technical “features”

- Pyrex needs a more active community.
- It could benefit from more of a shared-load development model.
  - Someone should do C++ support!
  - Someone should hack for optimization!
  - Somone should integrate with Jython/JNI and IronPython/CLI
  - Someone should port the Python stdlib.
  - …and so forth.
- Pyrex needs more marketing.

Discussion

- What is Pyrex's future?
- Should some of Python core be coded in Pyrex?
- How will Pyrex relate to other Python implementations?
- How should the Pyrex community self-organize?