PyD Documentation

Release 1.0

Kirk McDonald

May 26, 2017

Contents

Contents:

CHAPTER 1

Extending Python with D

PyD supports building python modules with D.

Basics

A minimal working PyD module looks something like

```
// A minimal "hello world" Pyd module.
module hello;

import pyd.pyd;
import std.stdio;

void hello() {
    writefln("Hello, world!");
}

extern(C) void PydMain() {
    def!(hello)();
    module_init();
}
```

This code imports the components of pyd, and provides the initializer function that python will call when it loads your module. It also exposes *hello* to python.

Some notes:

- *def* must be called before *module_init*. *class_wrap* must be called after *module_init*.
- PydMain will catch any thrown D exceptions and safely pass them to Python.

This extension is then built the usual way with distutils, except PyD provides some patches to support D compilers:

```
from pyd.support import setup, Extension
```

```
projName = 'hello'

setup(
    name=projName,
    version='0.1',
    ext_modules=[
        Extension(projName, ['hello.d'],
            extra_compile_args=['-w'],
            build_deimos=True,
            d_lump=True
        )
    ],
)
```

```
$ python setup.py install
```

Usage:

```
import hello
hello.hello()
```

CHAPTER 2

Embedding Python in D

A D program with embedded python might look like

```
module hello;

import std.stdio;
import pyd.pyd, pyd.embedded;

shared static this() {
    py_init();
}

void main() {
    writeln(py_eval!string("'1 + %s' % 2"));
}
```

Some Notes:

- you must call py_init to initialize the python interpreter before making any calls to python.
- *pyd.embedded* contains some goodies for calling python from D code
- it's even possible to expose D functions to python, then invoke them in python code in D code! (see examples/pyind)

Once again, we use distutils to compile this code using the special command *pydexe*:

```
from pyd.support import setup, Extension, pydexe_sanity_check

pydexe_sanity_check()
projName = 'hello'
setup(
    name=projName,
    version='1.0',
    ext_modules=[
        Extension(projName, ['hello.d'],
            build_deimos=True, d_lump=True
```

```
        )
    ],
)
```

```
$ python setup.py install
$ ./hello
1 + 2
```

InterpContext

One of the goodies in *pyd.embedded* is InterpContext - a class that wraps a python scope and provides some conveniences for data transfer:

```
module interpcontext;

import std.stdio;
import pyd.pyd, pyd.embedded;

shared static this() {
    py_init();
}

void main() {
    auto context = new InterpContext();
    context.a = 2;
    context.py_stmts("print ('1 + %s' % a)");
}
```

Miscellaneous

call stack is not deep enough

Certain python modules (i.e. `inspect`) expect python to have a nonempty call stack. This seems not to be the case in embedded python. To work around this, use `InterpContext.pushDummyFrame`:

```
context.pushDummyFrame();
py_stmts("import inspect");
context.popDummyFrame();
```

CHAPTER 3

PyD and distutils

PyD provides patches to distutils so it can use DMD, LDC, and GDC.

See also:

distutils

Mostly, this consists of a different `setup` that must be called, and a different `Extension` that must be used:

```
from pyd.support import setup, Extension

projName = 'hello'

setup(
    name=projName,
    version='0.1',
    ext_modules=[
        Extension(projName, ['hello.d'],
            extra_compile_args=['-w'],
            build_deimos=True,
            d_lump=True
        )
    ],
)
```

Command line flags

compiler

Specify the D compiler to use. Expects one of dmd, ldc, gdc.

```
python setup.py install --compiler=ldc
```

Default: dmd

debug

Have the D compiler compile things with debugging information.

```
python setup.py install --debug
```

Extension arguments

In addition to the arguments accepted by distutils' Extension, PyD's *Extension* accepts the following arguments:

version_flags

A list of strings passed to the D compiler as D version identifiers.

Default: `[]`

debug_flags

similar to version_flags for D debug identifiers

Default: `[]`

raw_only

When `True`, suppress linkage of all of PyD except the bare C API.

Equivalent to setting *with_pyd* and *with_main* to `False`.

Default: `False`

with_pyd

Setting this flag to `False` suppresses compilation and linkage of PyD. *with_main* effectively becomes `False` as well; *PydMain* won't be used unless PyD is in use.

Default: `True`

with_main

Setting this flag to `False` suppresses the use of *PydMain*, allowing the user to write a C-style init function instead.

Default: `True`

build_deimos

Build object files for deimos headers. Ideally, this should not be necessary; however some compilers (*cough* ldc) try to link to PyObject typeinfo. If you get link errors like

undefined symbol: _D6deimos6python12methodobject11PyMethodDef6__initZ

try setting this flag to `True`.

Default: `False`

optimize

Have D compilers do optimized compilation.

Default: `False`

d_lump

Lump compilation of all d files into a single command.

Default: `False`

d_unittest

Have D compilers generate unittest code

Default: `False`

d_property

Have D compilers enable property checks (i.e. trying to call functions without parens will result in an error)

Default: `False`

string_imports

Specify string import files to pass to D compilers. Takes a list of strings which are either paths to import files or paths to directories containing import files.

Default: `[]`

pydexe

PyD also provides a custom command to compile D code that embeds python. The format of setup.py stays the same.

```
from pyd.support import setup, Extension, pydexe_sanity_check

pydexe_sanity_check()
projName = 'pyind'
srcs = ['pyind.d']

setup(
    name=projName,
    version='1.0',
    ext_modules=[
    Extension(projName, srcs,
    build_deimos=True, d_lump=True
        )
    ],
)
```

Mixing C and D extensions

It is totally possible. Use PyD's `setup`.

```
from distutils.core import Extension as cExtension
from pyd.support import setup, Extension

module1 = Extension("x", sources = ['xclass.c'])
module2 = Extension("y", sources = ['hello.d'], build_deimos=True)

setup(
    name = "x",
    version = '1.0',
    description = "eat a taco",
    ext_modules = [
        module1,
        module2
    ]
);
```

CHAPTER 4

Type Conversion

PyD provides *d_to_python* and *python_to_d* for converting types to and from python. These functions almost always do a copy. If you want reference semantics, use *PydObject*.

D to Python

D Type	*Python Type*
bool	bool
Any integral type	bool
BigInt	long (int in python3)
float, double, real	float
std.complex.Complex	complex
std.datetime.Date	datetime.date
std.datetime.DateTime	datetime.datetime
std.datetime.SysTime	datetime.datetime
std.datetime.Time	datetime.time
string	str
dynamic array	list
static array	list
std.typecons.Tuple	tuple
associative array	dict
delegates or function pointers	callable object
a wrapped class	wrapped type
a wrapped struct	wrapped type
pointer to wrapped struct	wrapped type
PydObject	wrapped object's type
PyObject*	object's type

Python to D

Python Type	*D Type*
Any type	PyObject*, PydObject
Wrapped struct	Wrapped struct, pointer to wrapped struct
Wrapped class	Wrapped class
Any callable	delegate
array.array	dynamic or static array
Any iterable	dynamic or static array, PydInputRange
str	string or char[]
tuple	std.typecons.Tuple
complex	std.complex.Complex
float	float, double, real
int, long	Any integral type
bool	bool
buffer	dynamic or static array (with many dimensions!)
datetime.date	std.datetime.Date, std.datetime.DateTime, std.datetime.SysTime
datetime.datetime	std.datetime.Date, std.datetime.DateTime, std.datetime.SysTime, std.datetime.Time
datetime.time	std.datetime.Time

Numpy

Numpy arrays implement the buffer protocol, which PyD can efficiently convert to D arrays.

To convert a D array to a numpy array, use *pyd.extra.d_to_python_numpy_ndarray*.

Extending PyD's type conversion

PyD's type conversion can be extended using *ex_d_to_python* and *ex_python_to_d*. Each takes a delegate or function pointer that performs the conversion.

Extensions will only be used if PyD's regular type conversion mechanism fails. This would usually happen when an exposed function takes or returns an unwrapped class or struct.

```
module example;

import std.stdio;
import pyd.pyd;

struct S {
    int i;
}

S foo() {
    S s;
    s.i = 12;
    return s;
}

void bar(S s) {
    writeln(s);
}
```

```
extern(C) void PydMain() {
    ex_d_to_python((S s) => s.i);
    ex_python_to_d((int i) => S(i));

    def!foo();
    def!bar();
    module_init();
}
```

results:

```
example.foo()
example.bar(20)
```

```
12
S(20)
```

CHAPTER 5

Exposing D functions to python

The heart of PyD's function wrapping is the *def* template function.

```
import pyd.pyd;
import std.stdio;

void foo(int i) {
    writefln("You entered %s", i);
}

void bar(int i) {
    writefln("bar: i = %s", i);
}

void bar(string s) {
    writefln("bar: s = %s", s);
}

void baz(int i=10, string s="moo") {
    writefln("i = %s\ns = %s", i, s);
}

extern (C) void PydMain() {
    // Plain old function
    def!(foo)();
    // Wraps the lexically first function under the given name
    def!(bar, PyName!"bar1")();
    // Wraps the function of the specified type
    def!(bar, PyName!"bar2", void function(string))();
    // Wraps the function with default arguments
    def!(baz)();

    module_init();
}
```

Notes

- Any function whose return type and parameters types are convertible by
- All calls to *def* must occur before the call to module_init or py_init PyD's type conversion can be wrapped by *def*.
- *def* can't handle `out`, `ref`, or `lazy` parameters.
- *def* can't handle functions with c-style variadic arguments
- *def can* handle functions with default and typesafe variadic arguments
- *def* supports skipping default arguments (on the python side) and will automatically fill in any omitted default arguments
- *def*-wrapped functions can take keyword arguments in python

def-wrapped functions can be called in the following ways:

D function	Python call
`void foo(int i);`	foo(1) foo(i=1)
`void foo(int i = 2, double d = 3.14);`	foo(1, 2.0) foo(d=2.0) foo()
`void foo(int[] i...);`	foo(1) foo(1,2,3) foo([1,2,3]) foo(i=[1,2,3])

def template arguments

Aside from the required function alias, *def* recognizes a number of poor-man keyword arguments, as well as a type specifier for the function alias.

```
def!(func, void function(int), ModuleName!"mymodl")();
```

Order is not significant for these optional arguments.

ModuleName

specify the module in which to inject the function

Default: ''

Docstring

Specify the docstring to associate with the function

Default: `''`

PyName

Specify the name that python will bind the function to

Default: the name of the exposed function

CHAPTER 6

Exposing D classes to python

The heart of PyD's class wrapping features is the *class_wrap* function template.

Member wrapping

Python member type	D member type	PyD Param
instance function	instance function	Def!(Foo.fun)
static function	static function	StaticDef!(Foo.fun)
property	instance function or property	Property!(Foo.fun)
instance field	instance field	Member!(fieldname)
constructor	constructor	Init!(Args...)

Notes

- *Def* and *StaticDef* behave very much like *def*
- *Init* doesn't take named parameters
- *Member* takes a string, not an alias

Def template arguments

Docstring

See *Docstring*

PyName

See *PyName*

StaticDef template arguments

Docstring

See *Docstring*

PyName

See *PyName*

Property template arguments

Docstring

See *Docstring*

PyName

See *PyName*

Mode

Specify whether property is read-only, write-only, or read-write.

Possible values: `"r"`, `"w"`, `"rw"`, `""`

When `""`, determine mode based on availability of getter and setter forms.

Default: `""`

Member template arguments

Mode

See *Mode*

Operator Overloading

Operator	D function	PyD Param
`+ - * / % ^^ << >> & ^ | ~`	opBinary!(op)	OpBinary!(op)
`+ - * / % ^^ << >> & ^ | ~ in`	opBinaryRight!(op)	OpBinaryRight!(op)
`+= -= *= /= %= ^^= <<= >>= &= ^= |= ~=`	opOpAssign!(op)	OpAssign!(op)
`+ - ~`	opUnary!(op)	OpUnary!(op)
`< <= > >=`	opCmp	OpCompare!()
`a[i]`	opIndex	OpIndex!()
`a[i] = b`	opIndexAssign	OpIndexAssign!()
`a[i .. j]` (python `a[i:j]`)	opSlice	OpSlice!()
`a[i .. j] = b` (python `a[i:j] = b`)	opSliceAssign	OpSliceAssign!()
`a(args)`	opCall	OpCall!(Args)
`a.length` (python `len(a)`)	length	Len!()

Notes on wrapped operators

- only one overload is permitted per operator; however OpBinary and OpBinaryRight may "share" an operator.
- PyD only supports opSlice, opSliceAssign if both of their two indices are implicitly convertable to Py_ssize_t. This is a limitation of the Python/C API. Note this means the zero-argument form of opSlice (`foo[]`) cannot be wrapped.
- `~`, `~=`: Python does not have a dedicated array concatenation operator. `+` is reused for this purpose. Therefore, odd behavior may result with classes that overload both `+` and `~`. The Python/C API does consider addition and concantenation to be distinct operations, though.
- `in`: Semantics vary slightly. In python, `in` is a containment test and retursn a bool. In D, by convention `in` is a lookup, returning a pointer or null. PyD will check the boolean result of a call to the overload and return that value to Python.

Iterator wrapping

A wrapped class can be make iterable in python by supplying defs with the python names:

- `__iter__`, which should return `this`.
- `next`, which should return the next item, or null to signal termination. Signature must be `PyObject* next()`.

Alternatively, you can supply a single `__iter__` that returns a Range.

CHAPTER 7

PydObject

PydObject wraps a PyObject*. It handles the python reference count for you, and generally provides seamless access to the python object.

```
import std.stdio;
import pyd.pyd, pyd.embedded;

void main() {
    py_init();

    PydObject random = py_eval("Random()", "random");
    random.method("seed", 234);
    int randomInt = random.randrange(1, 100).to_d!int();
    PydObject otherInt = random.randrange(200, 250);

    writeln("result: ", otherInt + randomInt);

    PydObject ints = py_eval("[randint(1, 9) for i in range(20)]", "random");

    write("[");
    foreach(num; ints) {
        write(num);
        write(", ");
    }
    writeln("]");
}
```

Notes:

- Due to some residual awkwardness with D's properties, member functions with zero or one arguments must be accessed through *method*, *method_unpack*, etc. Member functions with two or more arguments can be called directly.
- Calling a member function will result in another PydObject; call `to_d!T()` to convert it to a D object.
- PydObjects are callable

- PydObjects are iterable
- PydObjects support the usual operator overloading.

Buffer protocol

PydObject exposes a near-raw interface to the buffer protocol which can be used to e.g. read values from a numpy array without copying the entire thing into a D data structure.

CHAPTER 8

Exception Wrapping

The raw Python/C API has a protocol for allowing C extensions to use Python's exception mechanism. As a user of PyD, you should never have to deal with this protocol. Instead, use PyD's mechanisms for translating a Python exception into a D exception and vice versa.

handle_exception

check if a Python exception has been set, and if it has, throw a `PythonException`. Clear the Python error code.

exception_catcher

wrap a D delegate and set a Python error code if a D exception occurs. Returns a python-respected "invalid" value (null or -1), or the result of the delegate if nothing was thrown.

Notes

- If your code interfaces with python directly, you should probably wrap it with `exception_catcher` (uncaught D exceptions will crash the python interpreter).
- All wrapped functions, methods, constructors, etc, handle D and python exceptions already.

CHAPTER 9

Authors

PyD was originally written by Kirk McDonald.

CeleriD (the distutils patches) was originally written by David Rushby.

Other contributors:

- Deja Augustine
- Don Clugston
- Ellery Newcomer

Special thanks to Tomasz Stachowiak and Daniel Keep for providing now-defunct metaprogramming modules which were vital to the early development of Pyd.

CHAPTER 10

Overview

Pyd is a library that provides seamless interoperability between the D programming language and Python.

```
module pyind;

import std.stdio;
import pyd.pyd;
import deimos.python.Python: Py_ssize_t, Py_Initialize;
import pyd.embedded;

shared static this() {
    on_py_init({
            def!(knock, ModuleName!"office",
                Docstring!"a brain specialist works here")();
            add_module!(ModuleName!"office")();
    });
    py_init();

    wrap_class!(Gumby,
        Def!(Gumby.query),
        ModuleName!"office",
        Property!(Gumby.brain_status),
        Property!(Gumby.resolution, Mode!"r"),
    )();
}

void knock() {
    writeln("knock! knock! knock!");
    writeln("BAM! BAM! BAM!");
}

class Gumby {
    void query() {
        writeln("Are you a BRAIN SPECIALIST?");
    }
```

```
    string _status;
    void brain_status(string s) {
        _status = s;
    }
    string brain_status() {
        return _status;
    }

    string resolution() {
        return "Well, let's have a look at it, shall we Mr. Gumby?";
    }
}


void main() {
    // simple expressions can be evaluated
    int i = py_eval!int("1+2", "office");
    writeln(i);

    // functions can be defined in D and invoked in Python (see above)
    py_stmts(q"<
        knock()
    >", "office");

    // functions can be defined in Python and invoked in D
    alias py_def!(
        "def holler(a):
            return ' '.join(['Doctor!']*a)",
        "office",
        string function(int)) call_out;

    writeln(call_out(1));
    writeln(call_out(5));

    // classes can be defined in D and used in Python

    auto y = py_eval("Gumby()","office");
    y.method("query");

    // classes can be defined in Python and used in D
    py_stmts(q"<
        class X:
            def __init__(self):
                self.resolution = "NO!"
            def what(self):
                return "Yes, yes I am!"
    >", "office");

    auto x = py_eval("X()","office");
    writeln(x.resolution);
    writeln(x.method("what"));

    py_stmts(q"<
        y = Gumby();
        y.brain_status = "HURTS";
        print ("MY BRAIN %s" % y.brain_status)
        print (y.resolution)
    >","office");
```

```
}
```