# Contents

1 Extending Python with D ................................................. 3
   1.1 Basics ..................................................................... 3

2 Embedding Python in D ....................................................... 5
   2.1 InterpContext .......................................................... 6
   2.2 Miscellaneous .......................................................... 6

3 PyD and distutils ............................................................... 7
   3.1 Command line flags ...................................................... 7
   3.2 Extension arguments ..................................................... 8
   3.3 pydexe ..................................................................... 9
   3.4 Mixing C and D extensions ............................................. 10

4 Type Conversion ................................................................. 11
   4.1 D to Python ............................................................. 11
   4.2 Python to D ............................................................. 12
   4.3 Numpy ................................................................... 12
   4.4 Extending PyD’s type conversion .................................... 12

5 Exposing D functions to python ............................................ 15
   5.1 def template arguments ............................................... 16

6 Exposing D classes to python ............................................... 19
   6.1 Member wrapping ...................................................... 19
   6.2 Def template arguments ............................................... 19
   6.3 StaticDef template arguments ....................................... 20
   6.4 Property template arguments ....................................... 20
   6.5 Member template arguments ....................................... 20
   6.6 Operator Overloading ............................................... 21
   6.7 Iterator wrapping ...................................................... 21

7 PydObject ..................................................................... 23
   7.1 Buffer protocol .......................................................... 24

8 Using Dub to build ............................................................ 25

9 Exception Wrapping .......................................................... 27
   9.1 handle_exception ........................................................ 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>exception_catcher</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>Authors</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>Overview</td>
<td>31</td>
</tr>
</tbody>
</table>
Contents:
Extending Python with D

PyD supports building python modules with D.

1.1 Basics

A minimal working PyD module looks something like

```d
// 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:
```python
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()
```
A D program with embedded python might look like

```d
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`:

```d
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
    ]

(continues on next page)
$ python setup.py install
$ ./hello
1 + 2

## 2.1 InterpContext

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

```d
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)");
}
```

## 2.2 Miscellaneous

### 2.2.1 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`:

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

```python
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
        ),
    ],
)
```

### 3.1 Command line flags

#### 3.1.1 compiler

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

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

Default: dmd
3.1.2 debug

Have the D compiler compile things with debugging information.

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

3.2 Extension arguments

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

3.2.1 version_flags

A list of strings passed to the D compiler as D version identifiers.
Default: `[]`

3.2.2 debug_flags

similar to version_flags for D debug identifiers
Default: `[]`

3.2.3 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`

3.2.4 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`

3.2.5 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`

3.2.6 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.
3.2.7 optimize

Have D compilers do optimized compilation.
Default: False

3.2.8 d_lump

Lump compilation of all d files into a single command.
Default: False

3.2.9 d_unittest

Have D compilers generate unittest code
Default: False

3.2.10 d_property

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

3.2.11 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: []

3.3 pydexe

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

```python
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
        )
    ],
)
```
3.4 Mixing C and D extensions

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

```python
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, d_lump=True)

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

### 4.1 D to Python

<table>
<thead>
<tr>
<th>D Type</th>
<th>Python Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>Any integral type</td>
<td>bool</td>
</tr>
<tr>
<td>BigInt</td>
<td>long (int in python3)</td>
</tr>
<tr>
<td>float, double, real</td>
<td>float</td>
</tr>
<tr>
<td>std.complex.Complex</td>
<td>complex</td>
</tr>
<tr>
<td>std.datetime.Date</td>
<td>datetime.date</td>
</tr>
<tr>
<td>std.datetime.Datetime</td>
<td>datetime.datetime</td>
</tr>
<tr>
<td>std.datetime.SysTime</td>
<td>datetime.datetime</td>
</tr>
<tr>
<td>std.datetime.Time</td>
<td>datetime.time</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
</tr>
<tr>
<td>dynamic array</td>
<td>list</td>
</tr>
<tr>
<td>static array</td>
<td>list</td>
</tr>
<tr>
<td>std.typecons.Tuple</td>
<td>tuple</td>
</tr>
<tr>
<td>associative array</td>
<td>dict</td>
</tr>
<tr>
<td>delegates or function pointers</td>
<td>callable object</td>
</tr>
<tr>
<td>a wrapped class</td>
<td>wrapped type</td>
</tr>
<tr>
<td>a wrapped struct</td>
<td>wrapped type</td>
</tr>
<tr>
<td>pointer to wrapped struct</td>
<td>wrapped type</td>
</tr>
<tr>
<td>PydObject</td>
<td>wrapped object’s type</td>
</tr>
<tr>
<td>PyObject*</td>
<td>object’s type</td>
</tr>
</tbody>
</table>
4.2 Python to D

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

4.3 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`.

4.4 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.

```d
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:

test.o

example.foo()
example.bar(20)

12
S(20)
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) {
    writeln("You entered %s", i);
}

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

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

void baz(int i=10, string s="moo") {
    writeln("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:

<table>
<thead>
<tr>
<th>D function</th>
<th>Python call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void foo(int i);</code></td>
<td><code>foo(1)</code></td>
</tr>
<tr>
<td><code>foo(i=1)</code></td>
<td></td>
</tr>
<tr>
<td><code>void foo(int i = 2, double d = 3.14);</code></td>
<td><code>foo(1, 2.0)</code></td>
</tr>
<tr>
<td><code>foo(d=2.0)</code></td>
<td></td>
</tr>
<tr>
<td><code>foo()</code></td>
<td></td>
</tr>
<tr>
<td><code>void foo(int[] i...);</code></td>
<td><code>foo(1)</code></td>
</tr>
<tr>
<td><code>foo(1,2,3)</code></td>
<td></td>
</tr>
<tr>
<td><code>foo([1,2,3])</code></td>
<td></td>
</tr>
<tr>
<td><code>foo(i=[1,2,3])</code></td>
<td></td>
</tr>
</tbody>
</table>

### 5.1 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.

```python
def!(func, void function(int), ModuleName!"mymod1")();
```

Order is not significant for these optional arguments.

#### 5.1.1 ModuleName

specify the module in which to inject the function.

Default: `'`
5.1.2 Docstring

Specify the docstring to associate with the function
Default: ''

5.1.3 PyName

Specify the name that python will bind the function to
Default: the name of the exposed function
Exposing D classes to python

The heart of PyD’s class wrapping features is the `class_wrap` function template.

### 6.1 Member wrapping

<table>
<thead>
<tr>
<th>Python member type</th>
<th>D member type</th>
<th>PyD Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance function</td>
<td>instance function</td>
<td>Def!(Foo.fun)</td>
</tr>
<tr>
<td>static function</td>
<td>static function</td>
<td>StaticDef!(Foo.fun)</td>
</tr>
<tr>
<td>property</td>
<td>instance function or property</td>
<td>Property!(Foo.fun)</td>
</tr>
<tr>
<td>instance field</td>
<td>instance field</td>
<td>Member!(fieldname)</td>
</tr>
<tr>
<td>constructor</td>
<td>constructor</td>
<td>Init!(Args...)</td>
</tr>
</tbody>
</table>

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

### 6.2 Def template arguments

#### 6.2.1 Docstring

See `Docstring`

#### 6.2.2 PyName

See `PyName`
6.3 StaticDef template arguments

6.3.1 Docstring

See Docstring

6.3.2 PyName

See PyName

6.4 Property template arguments

6.4.1 Docstring

See Docstring

6.4.2 PyName

See PyName

6.4.3 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: ""

6.5 Member template arguments

6.5.1 Mode

See Mode
6.6 Operator Overloading

<table>
<thead>
<tr>
<th>Operator</th>
<th>D function</th>
<th>PyD Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - * / % ^^ &lt;&lt; &gt;&gt; &amp; ^</td>
<td>~</td>
<td>opBinary!(op) OpBinary!(op)</td>
</tr>
<tr>
<td>+ - * / % ^^ &lt;&lt; &gt;&gt; &amp; ^</td>
<td>~ in</td>
<td>opBinaryRight!(op) OpBinaryRight!(op)</td>
</tr>
<tr>
<td>+= -= += /= %= ^^= &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
<td></td>
<td>= ~=</td>
</tr>
<tr>
<td>+ ~</td>
<td>opUnary!(op) OpUnary!(op)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>opCmp OpCompare!()</td>
<td></td>
</tr>
<tr>
<td>a[i]</td>
<td>opIndex OpIndex!()</td>
<td></td>
</tr>
<tr>
<td>a[i] = b</td>
<td>opIndexAssign OpIndexAssign!()</td>
<td></td>
</tr>
<tr>
<td>a[i .. j] (python a[i:j])</td>
<td>opSlice OpSlice!()</td>
<td></td>
</tr>
<tr>
<td>a[i .. j] = b (python a[i:j] = b)</td>
<td>opSliceAssign OpSliceAssign!()</td>
<td></td>
</tr>
<tr>
<td>a(args)</td>
<td>opCall OpCall!(Args)</td>
<td></td>
</tr>
<tr>
<td>a.length (python len(a))</td>
<td>length Len!()</td>
<td></td>
</tr>
</tbody>
</table>

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 convertible 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 concatenation to be distinct operations, though.

- in: Semantics vary slightly. In python, in is a containment test and returns 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.

6.7 Iterator wrapping

A wrapped class can be made 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.
PydObject wraps a PyObject*. It handles the python reference count for you, and generally provides seamless access to the python object.

```d
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.

7.1 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.
To use with dub, either specify the relevant subConfiguration for your python version, or run `source pyd_set_env_vars.sh <your python>` on linux or `pyd_set_env_vars.bat <your python>` on windows to set the relevant environment variables and use the `env` subConfiguration.

These scripts can be run from any directory, but to facilitate using PyD as a dependency pulled from the dub registry you can run `dub run pyd:setup` to copy them to the current directory for use, e.g. given you are in the current directory of a package that depends on pyd, run `dub run pyd:setup` followed by `source pyd_set_env_vars.sh`, then build your package as normal.
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.

### 9.1 handle_exception

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

### 9.2 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.
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.
Pyd is a library that provides seamless interoperability between the D programming language and Python.

```d
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?");
  }
}
```

(continues on next page)
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"));