On snakes and elephants

Using Python inside PostgreSQL

Jan Urbański
wulczer@wulczer.org

New Relic

PyWaw Summit 2015, Warsaw, May 26

For those following at home

Getting the slides

```
$ wget http://wulczer.org/pywaw-summit.pdf
```

1. Introduction
   - Stored procedures
   - PostgreSQL's specifics
2. The PL/Python language
   - Implementation
   - Examples
3. Using PL/Python
   - Real-life applications
   - Best practices

Outline

1 Introduction
- Stored procedures
- PostgreSQL's specifics

2 The PL/Python language

3 Using PL/Python

What are stored procedures

- procedural code **callable from SQL**
- used to implement operations that are not easily expressed in SQL
- encapsulate business logic

Stored procedure examples

Calling stored procedures

```
SELECT purge_user_records(142);

SELECT lower(username) FROM users;

CREATE TRIGGER notify_user_trig
    AFTER UPDATE ON users
    EXECUTE PROCEDURE notify_user();
```

Stored procedure languages

- most RDBMS have **one blessed language** in which stored procedures can we written
  - Oracle has PL/SQL
  - MS SQL Server has T-SQL
- but Postgres is **better**

Stored procedures in Postgres

- a stored procedure in Postgres is
  - information about input and output types
  - metadata like the name, the owner, additional modifiers
  - finally, **a bunch of text**
- a procedural language in Postgres is just a C extension module exposing a **single function**
- its job is to **execute** that piece of text, accepting and producing the perscribed types

Outline

1 Introduction
- Stored procedures
- PostgreSQL's specifics

2 The PL/Python language

3 Using PL/Python

Extensibility is king

- stored procedures in Postgres can be written in **any language**...
- ... as long as a handler has been defined for it
- several languages are officially supported
  - PL/pgSQL, a PL/SQL look-alike
  - PL/Tcp, PL/Perl and **PL/Python**
- and there's a lot of unofficial ones
  - Ruby, Lua, PHP, Java, Scheme, V8, R...

Trusted vs untrusted languages

- once installed, **trusted** languages are available to all users
  - for example, PL/pgSQL or PL/V8
- they need to provide a sandboxed execution environment for **arbitrary user code**
- the ability to create **untrusted** language functions is limited to database **superusers**

Outline

1 Introduction

2 The PL/Python language

- Implementation
- Examples

3 Using PL/Python

What PL/Python actually is

- the ability to run a **Python interpreter** inside the backend
- runs as the backend's OS user, so **untrusted**
- can run arbitrary Python code, including doing very **nasty** or really **crazy** things

What PL/Python actually is

- the ability to run a **Python interpreter** inside the backend
- runs as the backend's OS user, so **untrusted**
- can run arbitrary Python code, including doing very **nasty** or really **crazy** things
- but that's the **fun** of it!

How does it work

- the first time a PL/Python function is run, a **Python interpreter** is initialised inside the backend process
  - **preload** `plpython.so` to avoid initial slowdown
  - use **long-lived connections** to only pay the overhead once
- **Postgres types** are transformed into **Python types** and vice versa
  - only works for built-in types, the rest gets passed using the **string representation**

How does it work cd.

- function arguments are visible as **global variables**
- the function has access to various **magic globals** that describe the execution environment
  - the `plpy` module providing database access and utility functions
  - a dictionary with the old and new tuples if called as a trigger
  - dictionaries kept in memory between queries, useful for **caches**
- the module path depends on the server process's PYTHONPATH

Outline

1 Introduction

2 The PL/Python language

- Implementation
- Examples

3 Using PL/Python

PL/Python examples

Using Python modules

```
create function histogram(a float[], bins int = 10)
    returns int[]
as $$
    import numpy
    return numpy.histogram(a, bins)[0]
$$ language plpythonu;
```

PL/Python examples

Using Python modules cd.

```
create function get_host(url text) returns text as $$
import urlparse

return urlparse.urlparse(url).netloc
$$ language plpythonu;
```

PL/Python utility functions

Utility functions

```
create function find_table(name text)
    returns text[] as $$
import difflib

sql = 'select tablename from pg_tables'
result = plpy.execute(sql)
all_names = [table['tablename'] for table in result]

return difflib.get_close_matches(name, all_names)
$$ language plpythonu;
```

PL/Python utility functions

Utility functions cd.

```
create function add_unique_user(name text, email text)
    returns text as $$
lname, lemail = name, email
plan = plpy.prepare(
    'insert into users(name, email) values ($1, $2)',
    ('text', 'text'))

while True:
    try: plpy.execute(plan, (lname, lemail))
    except plpy.spiexceptions.UniqueViolation:
        lname = lname + '_'
    else: return lname
$$ language plpythonu;
```

PL/Python global dictionaries

Global dictionaries

```
create function get_mx(domain text) returns text as $$
import DNS, time

mx, expires = GD.get(domain, (None, 0))
if mx and time.time() < expires:
    return mx

GD[domain] = DNS.mxlookup(domain)[0][1], time.time() + 5
return GD[domain][0]
$$ language plpythonu;
```

PL/Python advanced examples

Avanced examples

```
create function check_mx() returns trigger as $$
import DNS

domain = TD['new']['email'].split('@', 1)[1]
try:
    DNS.mxlookup(domain) or plpy.error('no mx')
except DNS.ServerError:
    plpy.error('lookup failed for domain %s' % domain)
$$ language plpythonu;
```

PL/Python advanced examples

Avanced examples cd.

```
create function schedule(source text,
                         summary out text, location out text,
                         start out timestamptz)
    returns setof record as $$
import icalendar, requests

resp = requests.get(source)
cal = icalendar.Calendar.from_ical(resp.content)
for event in cal.walk('VEVENT'):
    yield (event['SUMMARY'], event['LOCATION'],
           event['DTSTART'].dt.isoformat())
$$ language plpythonu;
```

Outline

1 Introduction

2 The PL/Python language

3 Using PL/Python

- Real-life applications
- Best practices

Where to use PL/Python

- writing a particular piece of logic in a **nicer language** than PL/pgSQL
- doing **numerical computations** in the database with NumPy
- doing **text analysis** with NLTK
- writing a **constraint** that checks if a column contains JSON
  - or a protobuf stream
  - or a PNG image

Crazier ideas

- implementing a simple **cache layer** right in the database
- connecting to other Postgres instances and doing things to them
- communicating with **external services** to invalidate caches or trigger actions

Multicorn

- not really PL/Python, but **similar idea** under the hood
- uses the `foreign data wrapper` mechanism
  - foreign data wrappers are a way to present data residing in other storages as if they were local tables
- can use arbitrary Python code to provide **unified access** to disparate sources

Multicorn example

Filesystem access

```
create foreign table python_fs (
    package text, module text, content bytea)
server filesystem_srv options (
    root_dir '/usr/lib/python2.7/dist-packages',
    pattern '{package}/{module}.py',
    content_column 'content');
```

Multicorn modules

- out of the box, Multicord provides modules for filesystem, IMAP, LDAP, SQLAlchemy and a few more
- but it's easy to **write your own**!
- perfect for prototyping production-grade foreign data wrappers

Outline

1 Introduction

2 The PL/Python language

3 Using PL/Python

- Real-life applications
- Best practices

Organising PL/Python code

- keep your PL/Python code in a **module**
- make all your SQL functions **two-liners**
- test the Python code by **mocking out** magic variables
- it's a sharp tool, **be careful**

Organising PL/Python code

- keep your PL/Python code in a **module**
- make all your SQL functions **two-liners**

```
create function the_func(arg1 text, arg2 text)
    returns integer as $$
from myapp.plpython import functions
return functions.the_func(locals())
$$ language plpythonu;
```

- test the Python code by **mocking out** magic variables
- it's a sharp tool, **be careful**

Organising PL/Python code

- keep your PL/Python code in a **module**
- make all your SQL functions **two-liners**
- test the Python code by **mocking out** magic variables
- it's a sharp tool, **be careful**

Questions?