On snakes and elephants

Using Python with and in PostgreSQL

Jan Urbański
j.urbanski@wulczer.org

Ducksboard

PGCon 2012, Ottawa, May 18

For those following at home

Getting the slides

```
$ wget http://wulczer.org/snakes-and-elephants.pdf
```

Try the code

```
$ mkvirutalenv pgcon
$ pip install psycopg2 ipython requests
$ createdb pgcon
$ psql pgcon -c 'create extension plpythonu'
```

1. The language
   - A quick glance
   - Choosing the version
2. Drivers
   - DB API 2.0
   - Overview of existing drivers
   - Psycopg2 features and examples
3. ORMs
   - Why would you even want one?
   - Django ORM
   - SQLAlchemy
4. PL/Python
   - Use Python straight from the database
   - Best practices
   - Tricks, skulduggery and black magic

Outline

The language

What is Python?

Python is an old, boring, enterprise technology.

The language

What is Python?

Python is an old, boring, enterprise technology.

- Java (released 1995), PHP (released 1995), Postgres95 (released 1995), Python (released 1991)

The language

What is Python?

Python is an old, boring, enterprise technology.

- Java (released 1995), PHP (released 1995), Postgres95 (released 1995), Python (released 1991)
- there should be one – and preferably only one – obvious way to do it

The language

What is Python?

Python is an old, boring, enterprise technology.

- Java (released 1995), PHP (released 1995), Postgres95 (released 1995), Python (released 1991)
- there should be one – and preferably only one – obvious way to do it
- used at Google, the NASA and everywhere from web development through scientific software to system administration tools

Problems with Python

- multiple existing Postgres drivers make it difficult to decide which one to use in a new project
- the same goes for ORMs and other higher-level libraries
- the Python 2 vs Python 3 mess

Outline

Use Python 2.7

Python 2 or Python 3

- even though Python 3 is the future, the future is still not here
- many resources, libraries and tutorials work with Python 2 only
- follow some basic rules to make your future transition painless
  - make sure you know when you're dealing with characters and when you're dealing with bytes
  - don't use deprecated syntax - it's usually much uglier, anyway!
- use a recent version of Python 2

Outline

What is DB API 2.0

- a common API for Python database drivers, also known as PEP 249
- been around since 2001, with small modifications since then
- a bare bones specification, the lowest common denominator of database APIs
- many drivers provide their own extensions
- even though DB people tend to hate it, it has proven useful over the years

Outline

Quite a few drivers

- pyPgSQL
- PyGreSQL
- bpgsql
- ocpgdb
- pgasync
- pg8000
- py-postgresql
- psycopg2
- ... and the fun continues!

Use Psycopg2

Driver categories

- C wrappers around libpq
  - fast
  - features like PGPORT, .pgpass, PGSERVICE, SSL modes all work out of the box
  - one less thing to get wrong
- FEBE protocol implementations in Python
  - work with other Python interpreters, like PyPy or Jython
  - can give much tighter control over the communication with the backend

pyPgSQL

- libpq wrapper
- provides a DB API 2.0 compatible module
- last release was in 2006

PyGreSQL

- libpq wrapper
- provides both its own interface and a DB API 2.0 compatible module
- last release was in 2009, but the project seems to be alive
- supports most PostgreSQL features when using its own interface (but for instance server-side cursors are unsupported)

bpgsql

- pure Python implementation
- provides most of the DB API 2.0 interface
- last activity was in 2009

ocpgdb

- a wrapper around ODBC
- last project activity was in February 2012
- use this is you're married to ODBC somehow

pgasync

- pure Python implementation as a Twisted protocol
- last release was in 2005
- interesting because it's one of the few drivers that has well integrated asynchronous features

pg8000

- pure Python implementation
- provides a DB API 2.0 interface, with some extensions (but for instance large objects and server-side cursors are unsupported)
- actively maintained
- might be useful if you can't depend on C Python extensions or want to use an interpreter that does not support them, like PyPy or Jython)

py-postgresql

- pure Python implementation, with optional optimisations in C
- provides its own interface as well as a DB API 2.0 compatible module
- actively maintained
- quite featureful, going beyond simple querying - supports COPY, LISTEN/NOTIFY, server-side cursors, advisory locks, etc
- Python 3 only

psycopg2

- libpq wrapper
- based on DB API 2.0 with own extensions for Postgres-specific things
- actively maintained
- big community, widespread usage, quite featureful
- runs on Python 2 and 3
- well-defined threading behaviour

Outline

DB API 2.0 usage example

Short example

```
import psycopg2

conn = psycopg2.connect('dbname=schemaverse sslmode=require')
cur = conn.cursor()
cur.execute('select attack, defense from my_ships')

for row in cur.fetchall():
    print('total power %s' % row[0] + row[1])

conn.close()
```

Parameter passing

- placeholders in query strings are substituted for parameters
- the only conversion argument in use is %s
- supports both positional and named parameter passing, but not mixed
- the arguments should always be a sequence, even if there's only one!

Parameter passing examples

Parameter passing

```
cur.execute("select name from persons where last = %s",
            ("O'Hara", ))

cur.execute("update persons set last = %(prefix)s "
            "|| ' ' || %(last)s where surname = %(last)s",
            {"last": "O'Hara", "prefix": "Ms"})

cur.mogrify('select %s + %s', (1, 2))
```

Advanced psycopg2 features

- Python lists are transformed into Postgres arrays, Python tuples into constructs suitable for usage with IN
- you can register your own typecasters for Python classes and for Postgres types, some are provided out of the box:
  - hstore
  - composite types
  - UUID
  - INET
- by providing custom cursor classes you can hook into the querying process
- provide asynchronous interface and integration with many async frameworks

Custom cursor class example

Using a custom cursor

```
import logging
from psycopg2 import extensions

log = logging.getLogger('queries')

class LoggingCursor(extensions.cursor):
    def execute(self, query, vars=None):
        log.info('%s', self.mogrify(query, vars))
        return extensions.cursor.execute(self, query, vars)

cur = conn.cursor(cursor_factory=LoggingCursor)
```

Custom typecaster example

Automatic hstore typecasting

```
from psycopg2 import extras

extras.register_hstore(None, globally=True, oid=16829)

cur.execute('select %s || %s', ({'key1': 'val1'},
                                 {'key2': 'val2'}))

INFO:queries:select hstore(ARRAY['key1'], ARRAY['val1']) ||
                    hstore(ARRAY['key2'], ARRAY['val2'])
```

Asynchronous usage example

Asynchronous notifies

```
import psycopg2
from psycopg2.extras import wait_select

conn = psycopg2.connect('dbname=schemaverse sslmode=require',
                        async=1)
wait_select(conn)
cur = conn.cursor()
cur.execute('LISTEN error_channel')
wait_select(conn)
while 1:
    wait_select(conn)
    while conn.notifies:
        print("notify: %s" % conn.notifies.pop().payload)
```

Asynchronous usage example cd.

Asynchronous internals

```
import select
from psycopg2 import extensions

def wait_select(conn):
    while 1:
        state = conn.poll()
        if state == extensions.POLL_OK:
            break
        elif state == extensions.POLL_READ:
            select.select([conn.fileno()], [], [])
        elif state == extensions.POLL_WRITE:
            select.select([], [conn.fileno()], [])
```

Transaction management

- transactions are started automatically, but you need to commit them manually
- you can change that by setting `connection.autocommit = True`
- always use `connection.commit()` or `.rollback()`, otherwise you might get open transactions left around
- use a recent version, lots of wrinkles have been smoothed lately

Missing features

- PQexecParams support (marooned on discussions about the interface)
- easy access to prepared statements (connected with the above)
- asynchronous support for COPY
- quoting of identifiers

Outline

Reasons to use an ORM

- keeping the whole application in one language
- application developers don't know SQL well enough
- for simple cases it can speed up development and help with migrating between schema versions
- your PM doesn't ask whether to use one, she asks which one to use

Use SQLAlchemy

Outline

Characteristics of the Django ORM

- wildly popular, most people will end up having contact with it
- provides a handy administration tool if you use it across the board
- integrates well with the rest of the framework, making things work seamlessly (until they break)
- has a bunch of shortcomings

Shortcomings of the Django ORM

- no support for multiple column primary keys
- little control over generated queries
- creates and tears down connections for every request
- no support for Postgres-specific datatypes
- no support for stored procedures
- for a long time some Django and Postgres combinations would leave open transactions hanging

Coping with the Django ORM

- use a connection pooler
- monitor long-running transactions, upgrade to the latest version of Django and psycopg2
- use psycopg2!
- be careful to assess the trade-offs of using Django without its ORM
  - the admin won't work
  - authentication is tied to the Django User model, so it won't work out of the box
  - expect to jump through some hoops, but it's doable
- for simple CRUD applications, the Django ORM can actually be very useful (think 80/20)

Outline

Overview of SQLAlchemy

- not just an ORM, but a complete SQL toolkit in Python
- comprises two main parts, the ORM and the Expression Language
- if you dig long enough, you can do almost everything
- a complex beast, but well worth taming

ORM done right

- define your models, tables and mappings separately
  - models are Plain Old Python objects, you can use them without mapping them to database entities
  - you can also use the Declarative mode, where you define both the application objects and the database entities in one go
- supports:
  - CHECK constraints
  - database-specific types and operators
  - cascading updates and deletes
  - schemas
  - ... are you drooling yet?
- it strives to get the "last 20%" right

Mapper vs declarative style

Declare the object

```
class Car(object):
    def __init__(self, plate_no, make, price):
        self.plate_no = plate_no
        self.make = make
        self.price = price
```

Mapper vs declarative style

Declare the mapping

```
import sqlalchemy as sa
from sqlalchemy import orm

metadata = sa.MetaData()
car = sa.Table(
    'car', metadata,
    sa.Column('plate_no', sa.Unicode(length=6),
              primary_key=True),
    sa.Column('make', sa.UnicodeText(),
              sa.ForeignKey('car_makes', deferrable=True)),
    sa.Column('price', sa.Numeric, nullable=False))

orm.mapper(Car, car)
```

Mapper vs declarative style cd.

Declarative style

```
import sqlalchemy as sa
from sqlalchemy.ext import declarative

Base = declarative.declarative_base()

class Car(Base):
    __tablename__ = 'car'
    plate_no = sa.Column(sa.Unicode(length=6),
                         primary_key=True)
    make = sa.Column(sa.UnicodeText(),
                     sa.ForeignKey('car_makes',
                                   deferrable=True))
    price = sa.Column(sa.Numeric, nullable=False)
```

It's all SQL

Declarative style

```
from sqlalchemy import schema

print(schema.CreateTable(car).compile())
"""
CREATE TABLE car (
        plate_no VARCHAR(6) NOT NULL,
        make TEXT,
        price NUMERIC NOT NULL,
        PRIMARY KEY (plate_no),
        FOREIGN KEY(make) REFERENCES car_makes (make)
            DEFERRABLE
)
"""
```

Expression language

- either let the ORM handle the querying or construct expressions by hand
- all SQL constructs are supported, but since it's Python it's more composable
- allows you to rewrite places where the ORM does it wrong in something that feels like SQL
- if there's an SQL that can't be generated, it's a bug!

Expression language example

Constructing an expression

```
exp = (car.update()
       .where(sa.sql.func.lower(car.c.make) == 'chevy')
       .values(price=car.c.price + 2))

print exp
"""
UPDATE car SET price=(car.price + :price_1)
WHERE lower(car.make) = :lower_1
"""
```

Advantages over plain SQL

- more modular, allowing for code reuse and delegation of concerns
  - have one part of the code generate the update values and other the clauses
  - document and test using standard Python approaches
- use SQLAlchemy as a query generation backend and send literal SQL somewhere else for execution
- put logic in mapped objects to save code
  - transparently encrypt a column using `pg_crypto`
  - add generated columns to the objects, like `first_name || last_name`

Sessions and connections

- all access to the database is done through a Session
- a Session keeps track of the objects that got modified until flushed and committed or rolled back
- Sessions check out and return database connections from a built-in pool
- you can keep Session objects around to maintain long-lived connections, but remember to always close them after you're done
  - typically, one Session per WSGI process
  - use `scoped_session` to have a thread-local Session that can be shared
  - ensure the thread's session is closed after request is done, for instance using a middleware
  - closing a session does not close the underlying connection!

Outline

What's PL/Python

- 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's PL/Python

- 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 the 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
  - there's an extension module to parse hstore's to and from Python dicts

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 containing SPI 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 postmaster's PYTHONPATH

Outline

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

Outline

Ideas for using PL/Python

- doing numerical computations in the database with NumPy
- writing a constraint that checks if a column contains JSON
  - or a protobuf stream
  - or a PNG image
- connecting to other Postgres instances and doing things to them
- communicating with external services to invalidate caches or trigger actions
- checking if an email field's domain has a valid MX record

PL/Python examples

Using Python modules

```
CREATE FUNCTION find_extension(extname TEXT)
    RETURNS TEXT[] as $$
import difflib

sql = 'select name from pg_available_extensions'
result = plpy.execute(sql)
names = [extension['name'] for extension in result]

return difflib.get_close_matches(extname, names)
$$ LANGUAGE plpythonu;
```

PL/Python examples

Using Python modules

```
CREATE FUNCTION check_mx()
    RETURNS TRIGGER as $$
from dns import resolver

domain = TD['new']['email'].split('@', 1)[1]

try:
    resolver.query(domain, 'MX')
except resolver.NoAnswer:
    plpy.error('no MX record for domain %s' % domain)
$$ LANGUAGE plpythonu;
```

PL/Python examples

PGCon schedule

```
CREATE FUNCTION schedule(summary OUT TEXT,
                         location OUT TEXT,
                         start OUT TIMESTAMPTZ)
    RETURNS SETOF RECORD as $$
import icalendar, requests

resp = requests.get(GD['url'])
cal = icalendar.Calendar.from_ical(resp.content)

for event in cal.walk('VEVENT'):
    yield (event['SUMMARY'], event['LOCATION'],
           event['DTSTART'].dt.isoformat())
$$ LANGUAGE plpythonu;
```

Questions?