PostgreSQL, Python, and Squid.

Christophe Pettus
PostgreSQL Experts, Inc.
Let’s Talk Squid.

• What is a squid, anyway?
• For our purposes, a squid has three attributes:
  • length — in centimeters.
  • number of tentacles.
  • weight — in kilograms.
And of course!

- We’re using PostgreSQL.
- We’re using Python.
- We’re using psycopg2.
class Squid(object):
    def __init__(self, length, tentacles, weight):
        self.length = length
        self.tentacles = tentacles
        self.weight = weight

    def __str__(self):
        return '(' + str(self.length) + ',' +
              str(self.tentacles) + ',' +
              str(self.weight) + ')

s = Squid(length=12.5, tentacles=4, weight=5.7)
CREATE TABLE squid (  
squid_key bigserial primary key,  
length float,  
tentacles integer,  
weight float,  
CHECK (tentacles BETWEEN 3 AND 32)  
);

And we do something like this.
And we write something like this.

cur.execute(""
    INSERT INTO squid VALUES(%s, %s, %s)
""", [s.length, s.tentacles, s.weight] )
cur.commit()
cur.execute(""
    SELECT length, tentacles, weight FROM squid
    WHERE squid_key=%s
""", [skey])

squid_row = cur.fetchone()

squid = Squid(length=squid_row[0],
    tentacles=squid_row[1],
    weight=squid_row[2])
And we’re done.

• Well, that was a short presentation.
• But now, we want two different tables with Squid in them.
• That’s OK, we just replicate the schema...
CREATE TABLE atlantic_squid (  
squid_key bigserial primary key,  
length float,  
tentacles integer,  
weight float,  
CHECK (tentacles BETWEEN 3 AND 32)  
);

CREATE TABLE pacific_squid  
(LIKE atlantic_squid INCLUDING ALL);
And then we write something like...

cur.execute(
    "INSERT INTO " + ocean + "_squid VALUES(%s, %s, %s)",
    [s.length, s.tentacles, s.weight] )
cur.commit()
And at this point, we think...

- Wait, PostgreSQL has types!
- Maybe we can use PostgreSQL’s custom type facility.
But then you think...

• Oh, only big packages like PostGIS do stuff like that.

• We have to write C and PL/pgSQL and probably Scheme and Erlang for all we know.

• And how about operators? And indexing?

• Not for the likes of us Python people.
You would be wrong!

- It’s easy to create custom types in PostgreSQL.
- You can use custom PostgreSQL types in your application without much nasty code.
- You can write functions in the PostgreSQL database in Python.
PostgreSQL Custom Types.

- PostgreSQL has an extensive type system.
- You can create your own types.
- High-level aggregate types (structures of existing types).
- Low-level C-language types.
- Not today.
Declaring an aggregate type.

- Any time you declare a table, you also get a type with the same name and same structure.
- You can also just create a type without creating a new table.
CREATE TYPE squid AS ( 
  length float,
  tentacles integer,
  weight float
);

Like this!
That’s great, but...

- How do we get that custom type into and out of Python?
- psycopg2 has facilities for going both directions.
- Once set up, it Just Works.
class Squid(object):

    #...

    def __conform__(self, protocol):
        if protocol is psycopg2.extensions.ISQLQuote:
            return self

    def getquoted(self):
        return "'" + str(self) + "'::squid"
• Implement `__conform__` and `getquoted`.

• `__conform__` returns the object that implements `getquoted`.

• You can just return `self`.

• `getquoted` returns the object converted into “SQL quoted format.”
What’s “SQL Quoted Format”?

• Generally, it’s just a string.
• Any internal quotes need to follow the SQL quoting conventions.
• Custom types are serialized into strings.
• Aggregate types are enclosed in parens, with fields separated by commas.
For squids, it’s easy.

- We just use the string representation, since there are no fields that might contain quotes.
- If there were, you could just call the appropriate getquoted method on them.
- We wrap the whole thing in SQL string quotes, and add a ‘::squid’ cast to it.
Other People’s Children Classes

• What if we didn’t write the class?
  • `psycopg2.extensions.register_adapter(class, adapter)`

• The adapter function takes the object, returns a object that implements `getquoted`.

• If the `str()` of the object is fine, you can use `AsIs` to just return that.
We can create a table like this...

```sql
CREATE TABLE squids (
    squid_key       bigserial primary key,
    a_squid         squid
);
```
... and insert into it like this!

s = Squid(length=12.5, tentacles=4, weight=5.7
cur.execute("INSERT INTO squids(a_squid) VALUES(%s)",
            [s,])
But how do we get the squids out?

• Need to write a cast function.
• Takes the string representation from the database, and returns the object.
• We then register that function with psycopg2.
def cast_squid(value, cur):
    if value is None:
        return None

    match_object = re.match(r'\((?P<length>[0-9.]+),(?P<tentacles>[0-9]+),
(?P<weight>[0-9.]+)\)', value)

    if match_object is None:
        return None

    length = float(match_object.group('length'))
    tentacles = int(match_object.group('tentacles'))
    weight = float(match_object.group('weight'))

    return Squid(length=length, tentacles=tentacles, weight=weight)
And then we register it.

SQUID = psycopg2.extensions.new_type((72007,), "squid", cast_squid)
psycopg2.extensions.register_type(SQUID)
Not so fast. 72007?

- That’s the OID for the Squid type in *this particular* PostgreSQL database.
- All database schema objects have an OID.
- It’s different for every database that we create that type in.
- Changes if you restore the database from a `pg_dump`.
How do we get it?

cur.execute("SELECT NULL::Squid")
squid_oid = cur.description[0][1]
  # Can be executed once and cached.
And now SELECT works.

```python
g>> cur.execute("SELECT a_squid FROM squids")
g>> s = cur.fetchone()[0]
g>> print s.__class__
g<class '__main__.Squid'>
```
OK, but…

• What happened to our CHECK constraint?
• We don’t want mutant squids getting into our database.
• We could write a trigger…
• … but we don’t want to write PL/pgSQL.
We don’t have to!

- PL/Python!
- We can write our triggers and other functions in Python.
- The functions run in the PostgreSQL backend just like any other server-side code.
Great! Sign me up!

- PL/Python isn’t part of a database by default.
- `CREATE LANGUAGE plpythonu;`
- The “U” means Untrusted.
- Can bypass PostgreSQL’s access control system.
- Only superusers can create functions.
It didn’t like that.

• If you are using a package, make sure you have installed the appropriate -contrib package.

• If you are building from source, make sure you build with the --with-python option.
• PostgreSQL supports both.
• “plpython2u” “plpython3u”
• “plpythonu” gets you Python 2 right now, but might get you Python 3 in the future.
• The far, far future.
CREATE OR REPLACE FUNCTION hello_world() RETURNS bool AS
$hello_world$
plpy.notice("Hello, squids of the world!")
return True

$hello_world$
  LANGUAGE plpythonu;
And called the same.

```sql
squidy=# select hello_world();
NOTICE: Hello, squids of the world!
CONTEXT: PL/Python function "hello_world"
   hello_world

----------
  t
(1 row)
```
• Don’t declare a function body; PL/Python wraps it for you.

• Can call any installed Python package, but:
  • Cannot directly call any other stored procedure, in any language.
  • Use the SPI for that.
  • Module plpy contains that stuff.
One tentacle at a time, please.

• The PostgreSQL backend is single-threaded.

• Do not spawn threads within your PL/Python function.

• If you break it, you get to keep all the pieces.
CREATE OR REPLACE FUNCTION squid_trigger() RETURNS trigger AS
$squid_trigger$
from plpy import spiexceptions

calamari = TD["new"]['a_squid'][1:-1].split(',,')
tentacles = int(calamari[1])

if tentacles > 32 or tentacles < 3:
    raise spiexceptions.CheckViolation

return "OK"
$squid_trigger$
language plpythonu;
Calamari appetizer.

- In the TD structure, composite types are their string representation.
- In parameters to non-trigger stored procedures, they are passed (more logically) as hashes.
CREATE CONSTRAINT TRIGGER squid_trigger
AFTER INSERT OR UPDATE OF a_squid ON squids
NOT DEFERRABLE
FOR EACH ROW EXECUTE PROCEDURE squid_trigger();
Eldritch Monstrosities Avoided.

squidy=# INSERT INTO squids(a_squid) VALUES( (100, 47, 4.5)::squid );
ERROR: spiexceptions.CheckViolation:
CONTEXT: Traceback (most recent call last):
  PL/Python function "squid_trigger", line 10, in <module>
    raise spiexceptions.CheckViolation
PL/Python function "squid_trigger"
The Null Squid Hypothesis.

- Row types have strange rules around NULL.
  - \((1.0, \text{NULL}, 1.0)\)::squid IS NULL;
    - True.
  - \((1.0, \text{NULL}, 1.0)\)::squid IS NOT NULL;
    - Also true!
- NULL is a never-ending source of delight.
Seq Scan on squids (cost=0.00..253093.09 rows=50000 width=53) (actual time=6.917..2590.863 rows=1012 loops=1)
  Filter: (((a_squid).length >= 100::double precision) AND ((a_squid).length <= 101::double precision))
  Rows Removed by Filter: 9998989
Total runtime: 2591.113 ms
Squid total ordering.

- Squids are ordered by length, and nothing else.
- That’s just how squids roll.
- Can we speed up searching?
- Yes! We can create B-Tree indexes on custom types.
CREATE OR REPLACE FUNCTION squid_comp (left squid, right squid)
    RETURNS int as $squid_comp$
    if left["length"] < right["length"]:
        return -1
    elif left["length"] > right["length"]:
        return 1
    else:
        return 0

$squid_comp$
    LANGUAGE plpythonu
    IMMUTABLE STRICT;
CREATE OR REPLACE FUNCTION squid_eq (left squid, right squid) RETURNS bool AS
    $squid_eq$
    return left["length"] == right["length"]
    $squid_eq$
    LANGUAGE plpythonu
    IMMUTABLE STRICT;
CREATE OPERATOR = (  
    LEFTARG = squid,  
    RIGHTARG = squid,  
    PROCEDURE = squid_eq,  
    COMMUTATOR = =,  
    NEGATOR = <>,  
    RESTRICT = eqsel,  
    JOIN = eqjoinsel,  
    HASHES, MERGES  
);
CREATE OPERATOR <= (  
    LEFTARG = squid,
    RIGHTARG = squid,
    PROCEDURE = squid_le,
    COMMUTATOR = >=,
    NEGATOR = >,
    RESTRICT = scalarltssel,
    JOIN = scalarltltjoinsel
);
CREATE OPERATOR CLASS squid_ops
    DEFAULT FOR TYPE squid USING btree AS
    OPERATOR 1 < ,
    OPERATOR 2 <= ,
    OPERATOR 3 = ,
    OPERATOR 4 >= ,
    OPERATOR 5 > ,
    FUNCTION 1 squid_comp(squid, squid);
And then, Squidex!

CREATE INDEX squidex ON squids(a_squid);
Jet Propulsion!

Bitmap Heap Scan on squids  
(cost=2176.56..113217.70 
rows=50000 width=53) (actual time=10.991..12.367 rows=1012 
loops=1)

  Recheck Cond: ((a_squid >= ROW(100::double precision, 4, 
100::double precision)::squid) AND (a_squid <= ROW(101::double 
precision, 4, 100::double precision)::squid))

  -> Bitmap Index Scan on squidex  
(cost=0.00..2164.06 
rows=50000 width=0) (actual time=10.866..10.866 rows=1012 
loops=1)

  Index Cond: ((a_squid >= ROW(100::double precision, 
4, 100::double precision)::squid) AND (a_squid <= 
ROW(101::double precision, 4, 100::double precision)::squid))

Total runtime: 12.463 ms
Thanks for all the seafood.

- We can implement a custom type in PostgreSQL that integrates nicely with a Python class.
- … without losing any database features.
- … and those types can even have custom operators and comparisons.
- … and their own indexes!
I’m allergic to shellfish.

• This works with lots of stuff.
  • Range types, citext…

• Any time you have an advanced attribute type that you want to adapt to Python.
  • Whether or not you defined the type.

• Not just for squid anymore!
Questions?
Thank you!

@xof
cpettus@pgexperts.com