Extending PostGIS with Python

An introduction to plpygis

Benjamin Trigona-Harany

bosth@alumni.sfu.ca // benjamin@planet.com
Is this a Python client for PostGIS?
Why Python for PostGIS?

PostGIS spatial SQL functions: `ST_Area`, `ST_MakePoint`, `ST_Intersects`, `ST_GeoHash` ...
Why Python for PostGIS?

PostGIS spatial SQL functions: ST_Area, ST_MakePoint, ST_Intersects, ST_GeoHash ...

Advantages of Python functions:

- Procedural code
- Network access
- Geospatial Python modules
- Foreign Data Wrappers (Multicorn)
Why Python for PostGIS?

Python module to facilitate writing functions for PostGIS.

- Reads and writes PostGIS geometries
- Is Pythonic
- Has no extra dependencies *
- Implements __geo_interface__

* Shapely is optional
There are reasons *not* to do any of this.
Why Python for PostGIS?

What is plpygis?

Why not use it?

The drawbacks of Python for PostGIS:

- Most benefits are subjective
- Objectively slow
- Requires server access, probably as root
- Definitely compromises your security
PostgreSQL supports writing functions in a variety of procedural languages

- PL/pgSQL
- PL/Perl
- PL/Tcl
- PL/Python (2 & 3)

Other languages are available: PL/R, PL/v8, PL/Lua ...
"Create" the language in the database:

CREATE LANGUAGE plpythonu;

Function definition:

CREATE FUNCTION pymax(a integer, b integer) RETURNS integer AS $$
    if a > b:
        return a
    else:
        return b
$$ LANGUAGE plpythonu;

Execution:

SELECT pymax(1,2);

```sql
pymax
-----
 2
(1 row)
```
Enter plpygis
plpygis in practice

plpygis handles mapping between PostGIS `geometry` and PL/Python:

```sql
CREATE FUNCTION geo_example(geom geometry) RETURNS geometry AS $$
    from plpygis import Geometry
    g = Geometry(geom)
    -- place code here --
    return g
$$ LANGUAGE plpythonu;
```

And just to check it works

```sql
SELECT geom = geo_example(geom) FROM countries LIMIT 1;
```

<table>
<thead>
<tr>
<th>t</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1/47</td>
</tr>
</tbody>
</table>
Basic plpygis usage:

```python
>>> from plpygis import Geometry
>>> g = Geometry(pg_geometry)
>>> print(g.type)
Point
>>> print(g.srid)
4326
>>> print(p.dimz)
False
>>> print(p.dimm)
False
>>> print(g.x, g.y)
48.4262302 -123.3942419
>>> g.z = 23
>>> print(p.dimz)
True
>>> print(g.geojson)
{"coordinates": [48.42623, -123.39424, 23], "type": "Point"}
```
plpygis in practice

Geometries can be constructed manually ...

```python
>>> from plpygis import LineString
>>> l = LineString([(0,0), (1,1), (2,2)], srid=3857)
>>> print l.type
LineString
>>> print g.srid
3857
>>> print p.dimz
False
>>> print p.dimm
False
>>> print len(l.vertices)
3
>>> print g.geojson
{'coordinates': [[0, 0], [1, 1], [2, 2]], 'type': 'LineString'}
```

Any instance created this way can be returned from a PL/Python function as a PostGIS geometry.
plpygis in practice

What's the largest polygon in a multipolygon?

Use Shapely to provide the area calculation function and Python's native `max`.

```sql
CREATE FUNCTION largest_poly(geom geometry) RETURNS geometry AS $$
from plpygis import Geometry

g = Geometry(geom)

if g.type == "Polygon":
    return g
elif g.type == "MultiPolygon":
    largest = max(g.shapely,
                   key=lambda polygon: polygon.area)
    return Geometry.from_shapely(largest)
else:
    return None
$$ LANGUAGE plpythonu;
```

Note that plpygis only parses the full geometry when access to the coordinates is actually needed:

```python
>>> print g.x, g.y
```
Most countries in this list are composed of just a single polygon.

A few, such as Argentina, are made up of more than one polygon but are dominated by the largest of them.

Antigua and Barbuda, however, is a country that has more than one part but there is much more balance.
To prove that Antigua and Barbuda is a nicely balanced country, we can take a quick look at the geometries:

```sql
SELECT show(geom) FROM countries WHERE name LIKE 'Antigua%';
```

(1 row)
plpygis in practice

Could we have written our analysis with just spatial SQL?

```sql
CREATE FUNCTION largest_poly_native(polygons geometry)
RETURNS geometry
AS $$
  WITH geoms AS (SELECT (ST_Dump(polygons)).geom AS geom)
  SELECT geom
  FROM geoms
  ORDER BY ST_Area(geom) DESC LIMIT 1;
$$
LANGUAGE sql;
```

Same results as the Python `largest_poly` version.

It is arguably harder to write, but that's subjective.
What was show?
plpygis in practice

show is a wrapper around gj2ascii ...

```sql
CREATE FUNCTION show(geom geometry) RETURNS text AS $$
  from gj2ascii import render
  from plpygis import Geometry
  g = Geometry(geom)
  return render(g)
$$ LANGUAGE plpythonu
```

Note that `__geo_interface__` comes in handy here for integrating between Python modules.
SELECT show(geom) FROM countries WHERE name LIKE 'Malta';
plpygis in practice

And showc for colour ...

```sql
CREATE FUNCTION showc(geom geometry)
RETURNS text
AS $$
from gj2ascii import render, style
from plpygis import Geometry
layer = render(Geometry(geom), char="@")
return style(layer, stylemap={"@" : "green"})
$$ LANGUAGE plpythonu

SELECT name, showc(geom) FROM countries WHERE name = 'Ukraine';
```
What else makes sense in PL/Python? External services!

Let's geocode some points with `geopy` ...

```sql
CREATE OR REPLACE FUNCTION geocode(centroid geometry)
RETURNS text
AS $$
  from geopy import Nominatim
  from plpygis import Geometry
  p = Geometry(centroid)
  if p.type != "Point":
    return None
  nominatim = Nominatim()
  location = nominatim.reverse((p.y, p.x))
  return location.address
$$ LANGUAGE plpythonu;

SELECT name, geocode(ST_Centroid(geom))
FROM countries LIMIT 5;
```

<table>
<thead>
<tr>
<th>name</th>
<th>geocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aruba</td>
<td>Caya Lucas Wilfridus Juan Werleman, Santa Cruz, Aruba</td>
</tr>
<tr>
<td>Afghanistan</td>
<td>R, C, ولسوالي نلي, Daykundi, دايکندی, افغانستان</td>
</tr>
<tr>
<td>Angola</td>
<td>Ringoma, Bié, Angola</td>
</tr>
<tr>
<td>Albania</td>
<td>Bradashesh, Elbasan, Qarku i Elbasanit, 3001, Shqipëria</td>
</tr>
<tr>
<td>Antigua and Barb.</td>
<td>Hodges Bay, Antigua and Barbuda</td>
</tr>
</tbody>
</table>

(4 rows)
plpygis in practice

So why is plpygis a bad idea? Number one: speed ...

PL/Python

```sql
EXPLAIN ANALYZE SELECT largest_poly(geom)
FROM countries LIMIT 100;
```

```
QUERY PLAN

Seq Scan on countries (cost=0.00..122.30 rows=255 width=32)
Planning time: 0.036 ms
Execution time: 1176.503 ms
```

SQL

```sql
EXPLAIN ANALYZE SELECT largest_poly_native(geom)
FROM countries LIMIT 100;
```

```
QUERY PLAN

Seq Scan on countries (cost=0.00..122.30 rows=255 width=32)
Planning time: 0.337 ms
Execution time: 134.745 ms
```
So why is plpygis a bad idea? Number two: security ...

```
CREATE LANGUAGE plpythonu;
```

It's `plpythonu` and not `plpython` for a reason.

*PL/Python is only available as an "untrusted" language, meaning it does not offer any way of restricting what users can do in it and is therefore named `plpythonu`.*

Your PL/Python script is not sandboxed: it can do anything on your system with the permissions of the user running the database daemon (usually a user named `postgres`).
Some use cases where it *might* make sense to put PL/Python and plpygis:

- web services, either pulling data in or pushing it out
- with database triggers, when data is added gradually
- working with M dimensions
- writing data to the filesystem
Advanced plpygis
Advanced plpygis

`show` and `showc` take a single `geometry` parameter. This will show each country as a separate row ...

```
SELECT show(geom) FROM countries WHERE continent = 'Asia';
```

How can we pass in $n$ geometries to be rendered on a single map?

```
SELECT showall(geom) FROM countries WHERE continent = 'Asia';
```
Spatial aggregate functions
Advanced plpygis

SQL aggregate functions like `sum` or `ST_Collect` bring multiple rows' worth of data together.

They are defined by:

- "state transition function" (`SFUNC`) that keeps track as we handle each item and returns output (`STYPE`)

- "final function" (`FINALFUNC`) that creates the final output from the output (`STYPE`)

```sql
CREATE AGGREGATE showall(geometry) (  
    INITCOND='{}',  
    SFUNC=array_append,  
    STYPE=geometry[],  
    FINALFUNC=_final_geom_show  
);  
```
Advanced plygis

For `showall`, we don't need a special `sFUND`, we can use PostgreSQL's native `array_append`, which just adds each new item to an array.

We need `FINALFUNC`, which will take the array and render the geometries:

```sql
CREATE OR REPLACE FUNCTION _final_geom_show(geoms geometry[])
RETURNS text
AS $$
from gj2ascii import render_multiple
from plpygis import Geometry
from itertools import cycle
chars = [chr(i) for i in range(33,126)]
geojsons = [Geometry(g) for g in geoms]
layers = zip(geojsons, chars)
return render_multiple(layers, width)
$$ LANGUAGE plpythonu
```

`geometry[]` maps to a Python list type.
SELECT showall(geom) FROM countries WHERE continent = 'Asia';

(1 row)
Trigger functions
Triggers modify data as upon `INSERT, UPDATE or DELETE`.

```sql
CREATE TRIGGER add_city_geom BEFORE INSERT ON cities
FOR EACH ROW EXECUTE PROCEDURE _add_city_geom();

CREATE OR REPLACE FUNCTION _add_city_geom()
RETURNS TRIGGER AS $$
from plpygis import Point
from geopy import Nominatim

city = TD["new"]
if city["geom"] is None:
    geocoder = Nominatim()
    name = "{}", "{}", "{}".format(
        city["name"],
        city["adm1name"],
        city["adm0name"])
    location = geocoder.geocode(name)
city["geom"] = Point((location.longitude, location.latitude))
    city["geom"].srid = 4326
    return "MODIFY"
else:
    return "OK"
$$ LANGUAGE plpythonu;
```
Advanced plygis

```
SELECT name, adm1name, ST_AsText(geom)
FROM cities WHERE name = 'London';
```

```
<table>
<thead>
<tr>
<th>name</th>
<th>adm1name</th>
<th>st_astext</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>Kentucky</td>
<td>POINT(-84.083308264 37.128882262)</td>
</tr>
<tr>
<td>London</td>
<td>Westminster</td>
<td>POINT(-0.11866475932 51.501940588)</td>
</tr>
</tbody>
</table>
```

(2 rows)

Add a new London ...

```
INSERT INTO cities (name, adm1name)
VALUES ('London', 'Ontario');
```

and let the geometry be populated:

```
SELECT name, adm1name, ST_AsText(geom)
FROM cities WHERE name = 'London';
```

```
<table>
<thead>
<tr>
<th>name</th>
<th>adm1name</th>
<th>st_astext</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>Kentucky</td>
<td>POINT(-84.083308264 37.128882262)</td>
</tr>
<tr>
<td>London</td>
<td>Westminster</td>
<td>POINT(-0.11866475932 51.501940588)</td>
</tr>
<tr>
<td>London</td>
<td>Ontario</td>
<td>POINT(-81.249986654 42.969992404)</td>
</tr>
</tbody>
</table>
```

(3 rows)
Foreign data wrappers
Advanced plpygis

A foreign data wrapper (FDW) exposes remote objects as PostgreSQL tables:

- tables from another database
- email from IMAP

These three projects make spatial FDWs in Python possible:

- Multicorn
- geofdw
- plpygis

Note that the `pgsql-ogr-fdw` project already does spatial FDWs using GDAL!
Create a single "server" for all geocoding tables:

```
CREATE SERVER geocode
    FOREIGN DATA WRAPPER multicorn
    OPTIONS (wrapper 'geofdw.FGeocode');
```

Create two tables, one using the GoogleV3 geocoder and one using Nominatim:

```
CREATE FOREIGN TABLE fgc_google
    (rank INTEGER, address TEXT, geom geometry, query TEXT)
    SERVER geocode OPTIONS (service 'googlev3');

CREATE FOREIGN TABLE fgc_nominatim
    (rank INTEGER, address TEXT, geom geometry, query TEXT)
    SERVER geocode OPTIONS (service 'nominatim');
```

`fgc_google` and `fgc_nominatim` are now "virtual" tables with all known addresses.
Advanced plpygis

Select results from the geocoder matching our query string:

```sql
SELECT address, ST_AsText(geom) AS geom FROM fgc_google WHERE query = 'seaport hotel';
```

<table>
<thead>
<tr>
<th>address</th>
<th>geom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seaport Ln, Boston, MA 02210, USA</td>
<td>POINT Z (42.349255 -71.041385 0)</td>
</tr>
</tbody>
</table>

(1 row)

```sql
SELECT address FROM fgc_nominatim WHERE query = 'canada house';
```

<table>
<thead>
<tr>
<th>address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>High Commission of Canada, 5, Trafalgar Square, St. James's, Covent Garden, City of Westminster, London, England, UK</td>
<td></td>
</tr>
<tr>
<td>Canada House, West 54th Street, Diamond District, Manhattan, Manhattan Community Board 1, New York, New York, USA</td>
<td></td>
</tr>
<tr>
<td>Canada House, 29, Hampton Road, Cole Park, Strawberry Hill, Richmond-upon-Thames, London, England, UK</td>
<td></td>
</tr>
<tr>
<td>Canada House, The Circle, Southsea, Portsmouth, South East, England, UK</td>
<td></td>
</tr>
<tr>
<td>Canada House, Queen Victoria Way, Pirbright, Guildford, Surrey, South East, England, UK</td>
<td></td>
</tr>
<tr>
<td>Canada House, 28th Street, The Ministries, Juba, Central Equatoria, South Sudan</td>
<td></td>
</tr>
<tr>
<td>Canada House, Justine Close, Nabbingo, Wakiso, Central Region, Uganda</td>
<td></td>
</tr>
<tr>
<td>AerCap House, 65, St. Stephen's Green, Royal Exchange B ED, Dublin 2, Dublin, County Dublin, Ireland</td>
<td></td>
</tr>
</tbody>
</table>

(9 rows)
Advanced plpygis

Other FDWs using Python can interact with online datasets, local files, APIs, etc:

- Reverse geocode a point
- Search Planet's data base of imagery
- Expose a GeoJSON file online
- OSRM routing engine
- Web Feature Service
Genesis of plpygis
Given a table \texttt{countries} with columns \texttt{geom}, \texttt{name}, \texttt{pop_est} and so on, can we find out how PL/Python interprets PostGIS geometries?

```sql
CREATE FUNCTION geo_investigation(geom geometry)
RETURNS text
AS $$
    return geom
$$ LANGUAGE plpythonu;
```

```sql
SELECT name, geo_investigation(geom) FROM countries LIMIT 1;
```

<table>
<thead>
<tr>
<th>name</th>
<th>0106000020E6100000100000001030000000100000001A0...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aruba</td>
<td>0106000020E6100000100000001030000000100000001A0...</td>
</tr>
</tbody>
</table>

Hex-encoded Well-known Binary!
Making things spatial

Is the inverse true?

```sql
CREATE FUNCTION geo_investigation_ii()
RETURNS geometry
AS $$
    return "010100000000000000000000000000000000000000"
$$ LANGUAGE plpythonu;

SELECT ST_AsEWKT( geo_investigation_ii() );

st_asewkt
----------------------
POINT(0 0)
(1 row)
```

Observation #1: The bridge between PostGIS and PL/Python is the `geometry` type in PostgreSQL and Python's `str` type.
Observation #2: You don't need plpygis, but a) it makes your life easier and b) it's Pythonic.

```python
>>> from plpygis import Point
>>> p = Point((0, 1, 2))
>>> print p.srid
None
>>> print p.dimz
True
>>> print p.dimm
False
>>> print p.z
2
>>> print p.geojson
{"coordinates": [0, 1, 2], "type": "Point"}
```
Observation #2: You don't *need* plpygis, but a) it makes your life easier and b) it's Pythonic.

```python
>>> from plpygis import Point
>>> p = Point((0, 1, 2))
>>> print p.srid
None
>>> print p.dimz
True
>>> print p.dimm
False
>>> print p.z
2
>>> print p.geojson
{"coordinates": [0, 1, 2], "type": "Point"}
```
Making things spatial

It works the other way too.

```python
>>> from plpygis import Geometry
>>> g = Geometry("0101000020e6100000a5c810b68e364840a0cd60423bd95ec0")
>>> print g.type
Point
>>> print g.srid
4326
>>> print p.dimz
False
>>> print p.dimm
False
>>> print g.x, g.y
48.4262302 -123.3942419
>>> print g.geojson
{"coordinates": [48.4262302, -123.3942419], "type": "Point"}
```

Note that plpygis only parses the full WKB when access to the coordinates is actually needed:

```python
>>> print g.x, g.y
```
Project links

- plpygis: http://plpygis.readthedocs.io
- gj2ascii: https://pypi.python.org/pypi/gj2ascii
- Multicorn: http://multicorn.org
- geofdw: https://github.com/bosth/geofdw *

* Use master branch only
Slideshow created using remark.