# 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?

## What is plpygis?

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.

The drawbacks of Python for PostGIS:

- Most benefits are subjective
- Objectively slow
- Requires server access, probably as root
- Definitely compromises your security

# PL/* family

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 ...

# PL/* family

"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);
 pymax
-------
     2
(1 row)
```

# Enter plpygis

# plpygis in practice

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

```
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

```
SELECT geom = geo_example(geom) FROM countries LIMIT 1;
 ?column?
----------
 t
(1 row)
```

# plpygis in practice

Basic plpygis usage:

```
>>> 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 ...

```
>>> 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`.

```
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:

```
>>> print g.x, g.y
```

# plpygis in practice

```
SELECT name, ST_Area(largest_poly(geom)) / ST_Area(geom)
FROM countries LIMIT 10;
          name           |      ?column?
-------------------------+--------------------
 Aruba                   |                  1
 Afghanistan             |                  1
 Angola                  |  0.994655012831317
 Albania                 |                  1
 Andorra                 |                  1
 Antigua and Barb.       |  0.623786771016608
 Argentina               |  0.989333038974844
 Armenia                 |  0.998802877067866
 Bulgaria                |                  1
 Belarus                 |                  1
(10 rows)
```

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:

```
SELECT show(geom) FROM countries WHERE name LIKE 'Antigua%';
                          show
------------------------------------------------------------
                           +                               +
                       +   + +                             +
                       +   + + + + +                       +
                         +   + + + + +                     +
                         + +   + + + + +                   +
                         + +   + + + + +                   +
                         +   + + + + + + +                 +
                         + + + + + + + + +                 +
                         + + + + + + + + +                 +
                               + + + + + +                 +
                                   + + + +                 +
                                     + +                   +
                                                           +
                                                           +
                                                           +
                             + + +                         +
                           + + + + +                       +
                           + + + + +   + +                 +
                         + + + + + + + + +                 +
                     + + + + + + + + + + +                 +
                         + + + + + + + + + + + + +         +
                       + + + + + + + + + + + + +           +
                       + + + + + + + + + + + + + +         +
                     + + + + + + + + + + + + + +           +
                       + + + + + + + + + +                 +
                           + + +       + + +               +
(1 row)
```

# plpygis in practice

Could we have written our analysis with just spatial 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` ...

```
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';
                                      show
---------------------------------------------------------------------------------
         + + + + +                                                              +
 + + + + + + + + + + + +                                                        +
 + + + + + + + + + + + + + +                                                    +
 + + + + + + + + + + + + + + +                                                  +
   + + + + + + + + + + + + + +                                                  +
       + + + + + + + + +                                                        +
                                                                                +
                                                                                +
                                                                                +
                                 + + +                                          +
                               + + +                                            +
                               + + + + + + + + + + +                            +
                                 + + + + + + + + + + + +                        +
                                 + + + + + + + + + + + + +                      +
                                 + + + + + + + + + + + + + + +                  +
                                 + + + + + + + + + + + + + + + + + +            +
                                   + + + + + + + + + + + + + + + + +            +
                                   + + + + + + + + + + + + + + + + +            +
                                   + + + + + + + + + + + + + + + + + + +        +
                                   + + + + + + + + + + + + + + + + + + + + +    +
                                   + + + + + + + + + + + + + + + + + + + + + +  +
                                     + + + + + + + + + + + + + + + + + + + + + ++
                                       + + + + + + + + + + + + + + + + + + + + ++
                                         + + + + + + + + + + + + + + + + + + + ++
                                           + + + + + + + + + + + + + + + + + + ++
                                               + + + + + + + + + + + + + +      +
                                                     + + + + + + + + + +        +
                                                             + + + + + +        +

(1 row)
```

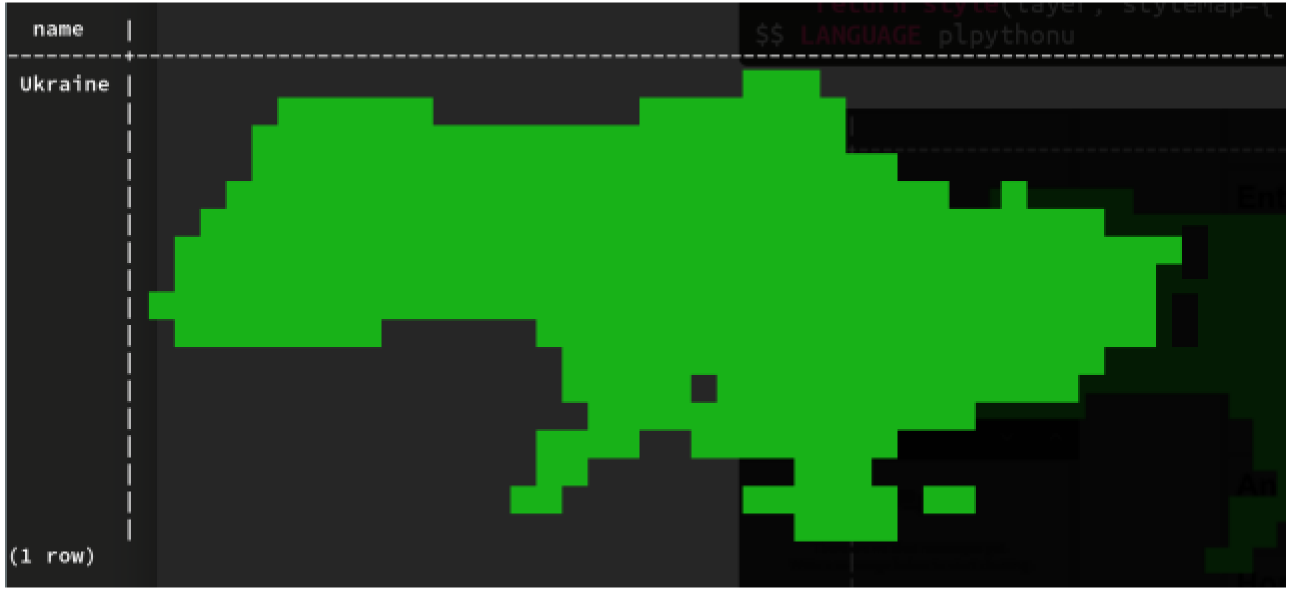
# plpygis in practice

## And `showc` for colour ...

```
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';
```



# plpygis in practice

What else makes sense in PL/Python? External services!

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

```
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;
       name           |                    geocode
----------------------+-------------------------------------------------------------
 Aruba                | Caya Lucas Wilfridus Juan Werleman, Santa Cruz, Aruba
 Afghanistan          | R-C, نیلی ولسوالی, Daykundi افغانستان ,دایکندی
 Angola               | Ringoma, Bié, Angola
 Albania              | Bradashesh, Elbasan, Qarku i Elbasanit, 3001, Shqipëria
 Antigua and Barb.    | Hodges Bay, Antigua and Barbuda
(4 rows)
```

# plpygis in practice

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

## PL/Python

```
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

```
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
```

# plpygis in practice

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`).

# plpygis in practice

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`)

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

# Advanced plpygis

For `showall`, we don't need a special `SFUNC`, 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:

```
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';
                                     showall
-----------------------------------------------------------------------------------
                        : : :                                                   +
                      : : : : : : :               E                 ( (         +
          : : : : : : : : : : : : : : : : E E E E E E E E E E E ( ( ( ( ( (     +
                : : : : : : : : : : : ( ( ( E E E E E E E E E E E ( ( ( ( ( (   +
              : : Y Y : : : : : : : ( ( ( ( ( ( E E E E E E E ( ( ( ( ( ( (     +
W     W 2       U U Y Y Y : Y . . . ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( K K     +
W W W W 5 5     U U U U Y Y T T ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( (       K       +
  R R 6 6 5 5 5 5 5 5 U ! ! ! G G ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( (     =       +
  R 6 6 6 5 5 5 5 5 5 ! ! ! G G , , ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( (             +
8 ; N 6 6 6 5 5 5 5 5 ! ! G G G , , ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( (           +
  N N N N N   5 5 5 5 G G G G , , , , D ( ( ( , , ( ( ( ( ( ( ( ( (             +
  N N N N N N     5 5 G G G , , , , , , , & , , C ( ( ( ( ( ( ( ( (             +
    N N N N N " " F         , , , , , , , , & C C C ( ( ( ( ( ( (               +
    N N N N N N N F           , , , , , ,     C C C ? ? Z                       +
      N N N [ F F             , , , , ,         C S ? S             J           +
        [ [ [                   , ,               S S S ?                       +
        [                       , ,               C S 0 0 Z                     +
                                  ,               S     Z         J             +
                                    A                                 J         +
                                                    H           H               +
                                                  +         H H +               +
                                                    +       + + +               +
                                                    + +     + +   +             +
                                                      +                         +
                                                          + +                   +
                                                                                +
(1 row)
```

# Trigger functions

# Advanced plpygis

Triggers modify data as upon `INSERT`, `UPDATE` or `DELETE`.

```
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 plpygis

```
SELECT name, adm1name, ST_AsText(geom)
FROM cities WHERE name = 'London';
  name  |  adm1name    |                st_astext
--------+--------------+-------------------------------------
 London | Kentucky     | POINT(-84.083308264 37.128882262)
 London | Westminster  | POINT(-0.11866475932 51.501940588)
(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';
  name  |  adm1name    |                st_astext
--------+--------------+-------------------------------------
 London | Kentucky     | POINT(-84.083308264 37.128882262)
 London | Westminster  | POINT(-0.11866475932 51.501940588)
 London | Ontario      | POINT(-81.249986654 42.969992404)
(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!

# Advanced plpygis

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:

```
SELECT address, ST_AsText(geom) AS geom FROM fgc_google WHERE query = 'seaport hotel';
               address                |                geom
--------------------------------------+-----------------------------------
 1 Seaport Ln, Boston, MA 02210, USA | POINT Z (42.349255 -71.041385 0)
(1 row)
```

```
SELECT address FROM fgc_nominatim WHERE query = 'canada house';
                                                                            address
--------------------------------------------------------------------------------------
 High Commission of Canada, 5, Trafalgar Square, St. James's, Covent Garden, City of We
 Canada House, West 54th Street, Diamond District, Manhattan, Manhattan Community Boarc
 Canada House, 29, Hampton Road, Cole Park, Strawberry Hill, Richmond-upon-Thames, Lonc
 Canada House, The Circle, Southsea, Portsmouth, South East, England, UK
 Canada House, Queen Victoria Way, Pirbright, Guildford, Surrey, South East, England, U
 Canada House, 28th Street, The Ministries, Juba, Central Equatoria, South Sudan
 Canada House, Justine Close, Nabbingo, Wakiso, Central Region, Uganda
 Aercap House, 65, St. Stephen's Green, Royal Exchange B ED, Dublin 2, Dublin, County D
 בית קנדה, 1, שבי ציון, רובע א', אשדוד, מחוז הדרום, מדינת ישראל
(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

# Making things spatial

Given a table `countries` with columns `geom`, `name`, `pop_est` and so on, can we find out how PL/Python interprets PostGIS geometries?

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

```
SELECT name, geo_investigation(geom) FROM countries LIMIT 1;
 name   |
-------+-----------------------------------------------------
 Aruba | 0106000020E61000000100000001030000000100000001A0...
```

Hex-encoded Well-known Binary!

# Making things spatial

Is the inverse true?

```
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.

# Making things spatial

Observation #2: You don't *need* plpygis, but a) it makes your life easier and b) it's Pythonic.

```
>>> 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

Observation #2: You don't *need* plpygis, but a) it makes your life easier and b) it's Pythonic.

```
>>> 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"}
>>> print p.wkb
"0101000080000000000000000000000000000000f03f0000000000000040"
```

# Making things spatial

It works the other way too.

```
>>> 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:

```
>>> 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.