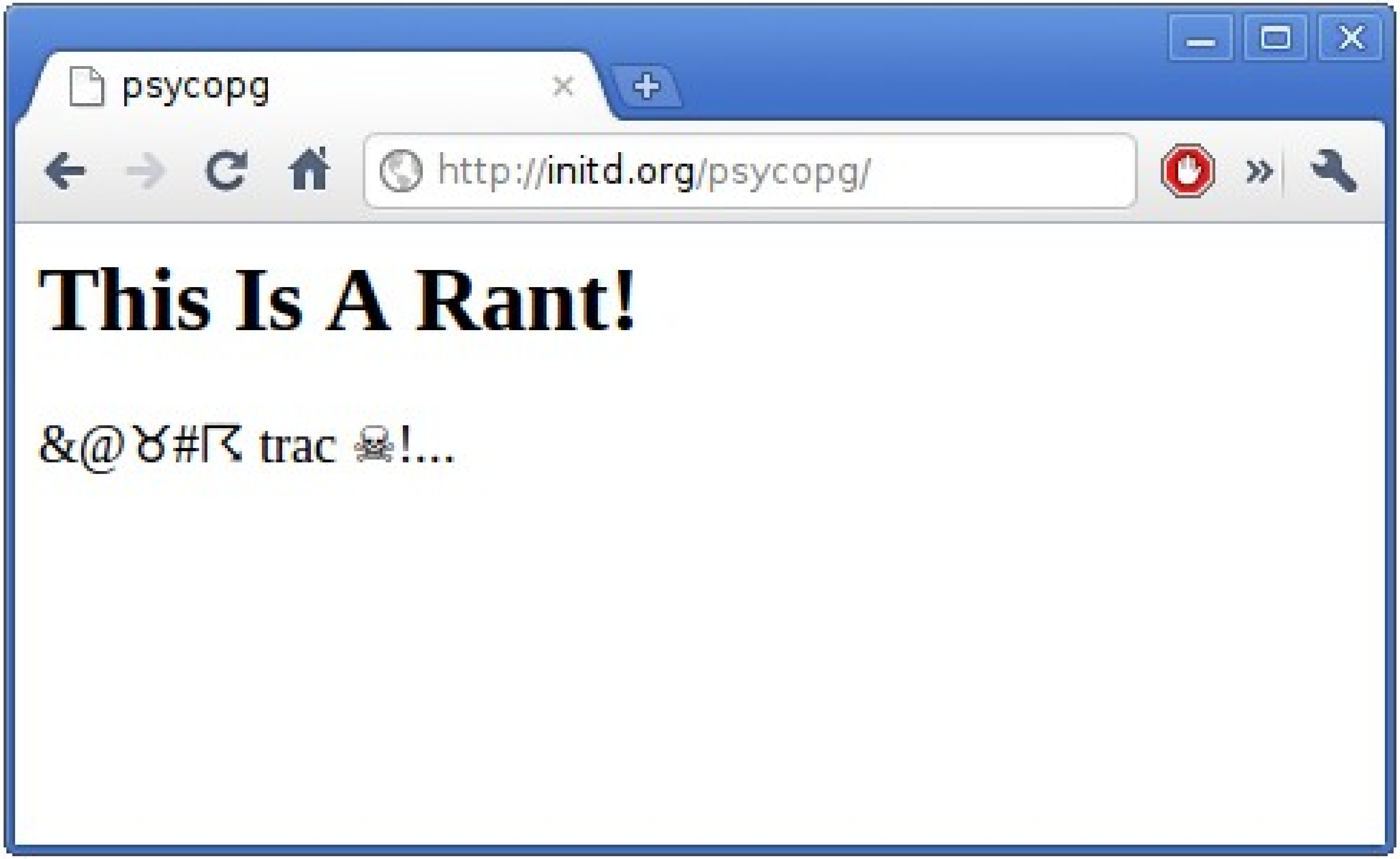
Advanced access to PostgreSQL from Python with **psycopg2**

“classic” psycopg homepage



This Is A Rant!

&@Ծ#☈ trac ☠!...

Psycopg characteristics

- LGPL license
- Written mostly in C
- libpq wrapper
  - Python 2.4 – 2.7
  - PostgreSQL >= 7.4
    - dropped V2 protocol support in 2.3
- Implements Python DB-API interface
  - `connection` wraps the session
  - `cursor` holds a result

Latest history

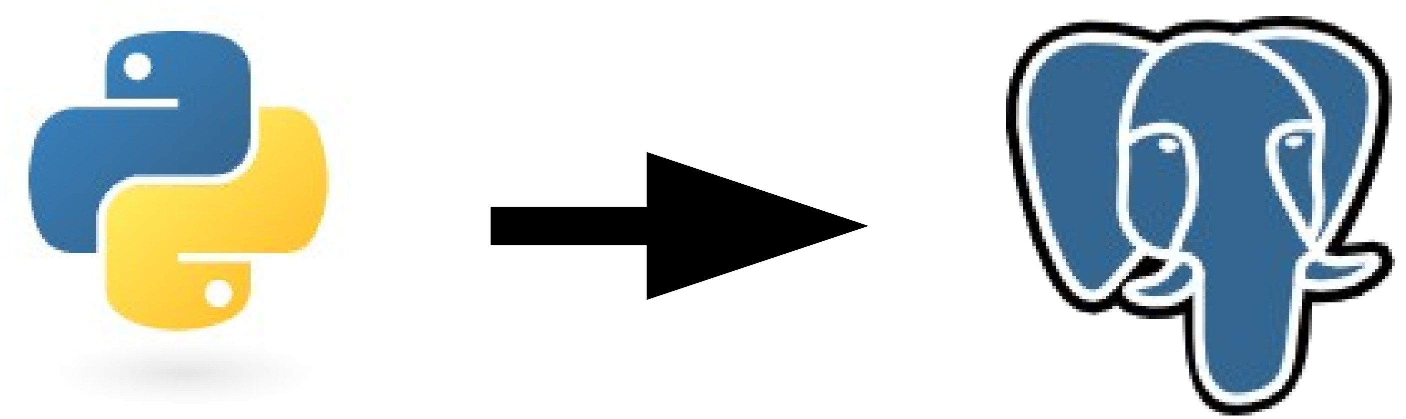
- Before 2010: a lot of undocumented features
  - Py-PG adaptation, SSC, notifies
- 2.2: async support
- 2.3: notify payload, 2PC, hstore

Let's talk about...

- Types adaptation
- Server-side cursors
- Transactions handling
- Async support
- Server notifications

- **Types adaptation**
- Server-side cursors
- Transactions handling
- Async support
- Server notifications

Python objects adaptation



- An *adapter* maps Python objects into SQL syntax
  - built-in adapters for basic objects/types
- Adapters are registered by type
  - since Psycopg 2.3: Liskov-friendly

Adapter example: XML

```
from xml.etree import cElementTree as ET
from psycopg2.extensions import \
     adapt, register_adapter

class ElementAdapter:
    def __init__(self, elem):
        self.elem = elem
    def getquoted(self):
        return "%s::xml" \
            % adapt(ET.tostring(elem))

register_adapter(type(ET.Element('')),
                 ElementAdapter)
```

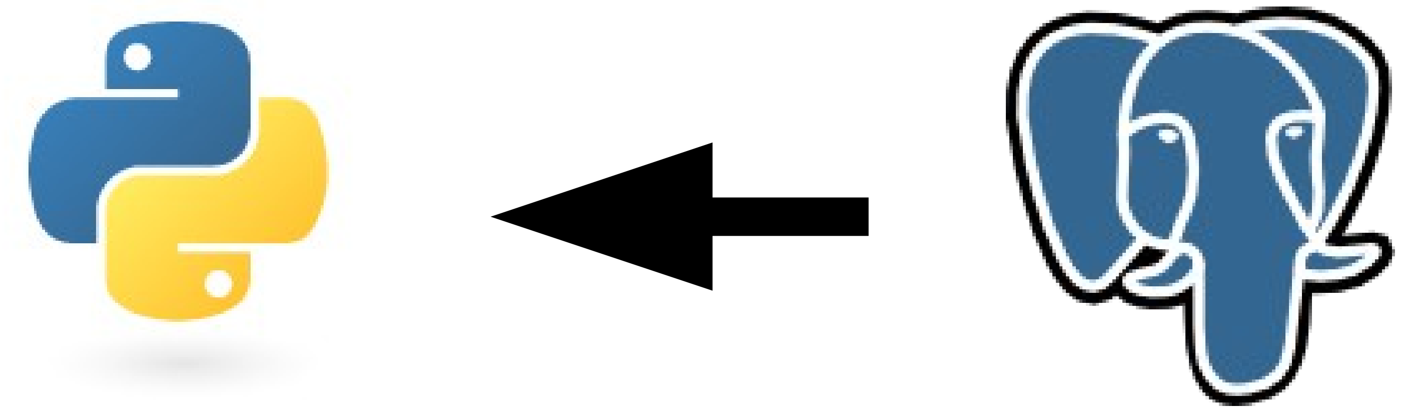
Adapter example: XML

```
elem = ET.fromstring(
    "<doc>Hello, 'xml'!</doc>")

print adapt(elem).getquoted()
# '<doc>Hello, ''xml''!</doc>'::xml

cur.execute("""
    INSERT INTO xmltest (xmldata)
    VALUES (%s);""", (elem,))
```

PostgreSQL types adaptation



- A *typecaster* maps PostgreSQL types into Python objects
- Typecasters are registered per oid
- Global, connection or cursor scope

Typecaster example: XML

```python
def cast_xml(value, cur):
    if value is None: return None
    return ET.fromstring(value)

from psycopg2.extensions import \
    new_type, register_type

XML = new_type((142,), "XML", cast_xml)
register_type(XML)
```

Typecaster example: XML

```
cur.execute("""
    SELECT xmldata FROM xmltest
    ORDER BY id DESC LIMIT 1;""")

elem = cur.fetchone()[0]

print elem.text
# Hello, 'xml'!
```

dict-hstore adaptation

- *hstore:* associative array of strings
  - `foo => bar, baz => whatever`
- Improved in PostgreSQL 9.0
  - capacity and indexing
- Adapter new in Psycopg 2.3
  - can deal with both pre-9.0 and 9.0 PostgreSQL

dict-hstore adaptation

```
psycopg2.extras.register_hstore(cnn)

cur.execute("SELECT 'a => b'::hstore;")
print cur.fetchone()[0]
# {'a': 'b'}

cur.execute("SELECT %s;",
    [{'foo': 'bar', 'baz': None}])
# SELECT hstore(ARRAY[E'foo', E'baz'],
#               ARRAY[E'bar', NULL])
```

hstore: **SO** useful

- ...if I only could remember the operators

```
cur.execute(                    # has a key?
    "select * from pets where data ? %s;",
    ('tail', ))
cur.execute(                    # has all keys?
    "select * from pets where data ?& %s;",
    (['tail', 'horns'], ))
cur.execute(                    # has any key?
    "select * from pets where data ?| %s;",
    (['wings', 'fins'], ))
cur.execute(                    # has keys/values?
    "select * from pets where data @> %s;",
    ({'eyes': 'red', 'teeth': 'yellow'}, ))
```

- Types adaptation
- **Server-side cursors**
- Transactions handling
- Async support
- Server notifications

Problem: out of memory

- I have this problem:

```
cursor.execute(
    "select * in big_table")
for record in cursor:
    whatever(record)
```

- Well, it doesn't work: “out of memory”!

Problem: out of memory

- `cursor.execute()` moves all the dataset to the client
  - `PGresult` structure
- `cursor.fetch*()` only manipulates client-side data
  - `PGresult` ➔ Python objects
- `DECLARE` to the rescue!

Named cursors

- `connection.cursor(`*`name`*`)`
- `cursor.execute(`*`sql`*`)`
  - ➔ `DECLARE` *`name`* `CURSOR FOR` *`sql`*
- `cursor.fetchone()`
  - ➔ `FETCH FORWARD 1 FROM` *`name`*
- `cursor.fetchmany(`*`n`*`)`
  - ➔ `FETCH FORWARD` *`n`* `FROM` *`name`*

Named cursor

- If you need to manipulate **many** records client-side
- Best strategy:

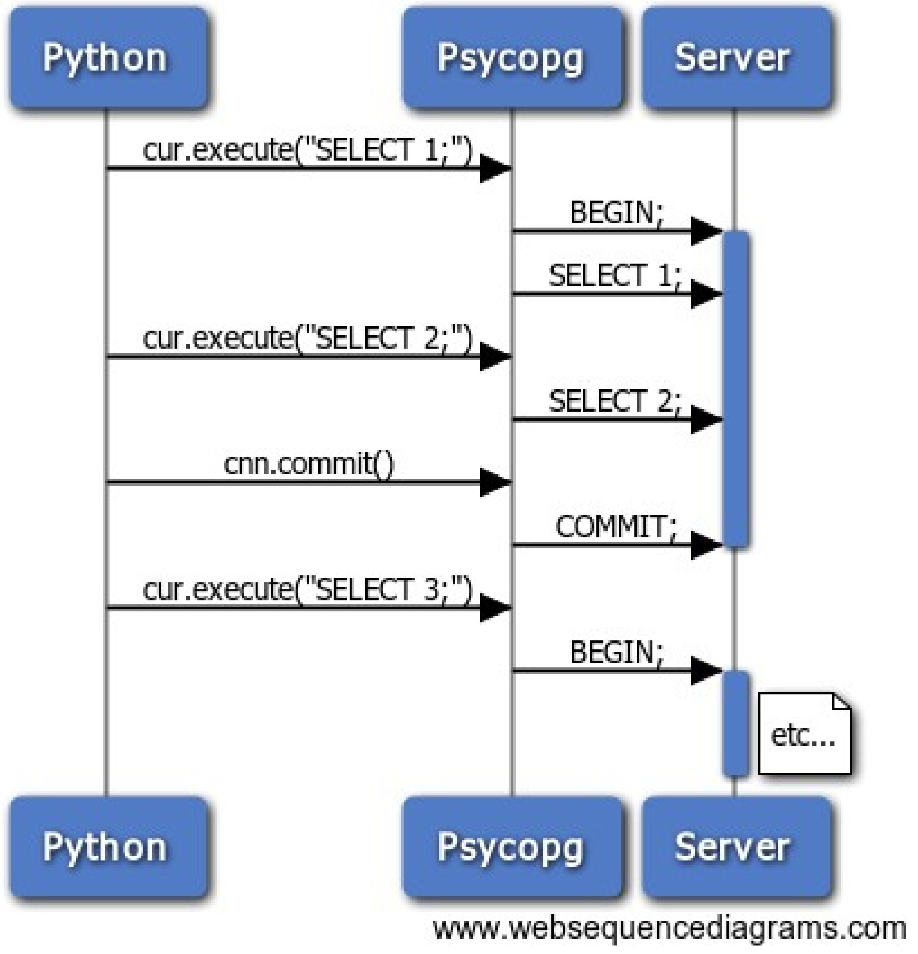
```
cur = connection.cursor(name)
cur.execute()
cur.fetchmany(n)
```

- Reasonable *n* to have good memory usage and not too many network requests

- Types adaptation
- Server-side cursors
- **Transactions handling**
- Async support
- Server notifications

Transactions handling

- The connection "has" the transaction
  - all its cursors share it
- Every operation in a transaction
  - DB-API requirement
- Until `.commit()` or `.rollback()` you are "<IDLE> in transaction"
  - bad for many reasons



Close that transaction!

- People are notoriously good at remembering boring details, aren't they?
  - `conn.commit()/conn.rollback()`
- Use a decorator/context manager

```
@with_connection
def do_some_job(conn, arg1, arg2):
    cur = conn.cursor()
    # ...
with get_connection() as conn:
    cur = conn.cursor()
    # ...
```

- Go *autocommit* if you need to

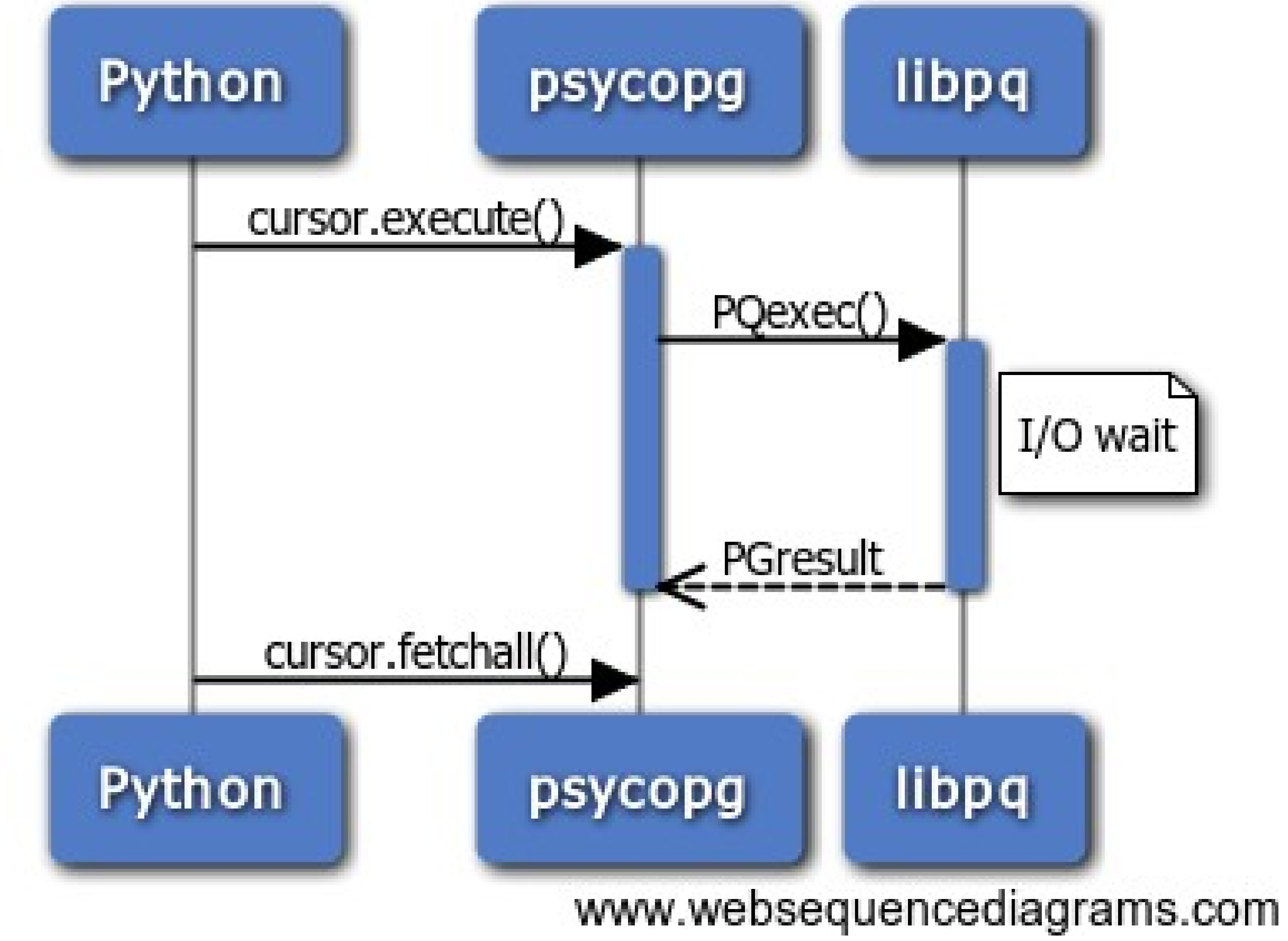
  -
    ```
    conn.set_isolation_level(
        ISOLATION_LEVEL_AUTOCOMMIT)
    ```

- Types adaptation
- Server-side cursors
- Transactions handling
- **Async support**
- Server notifications

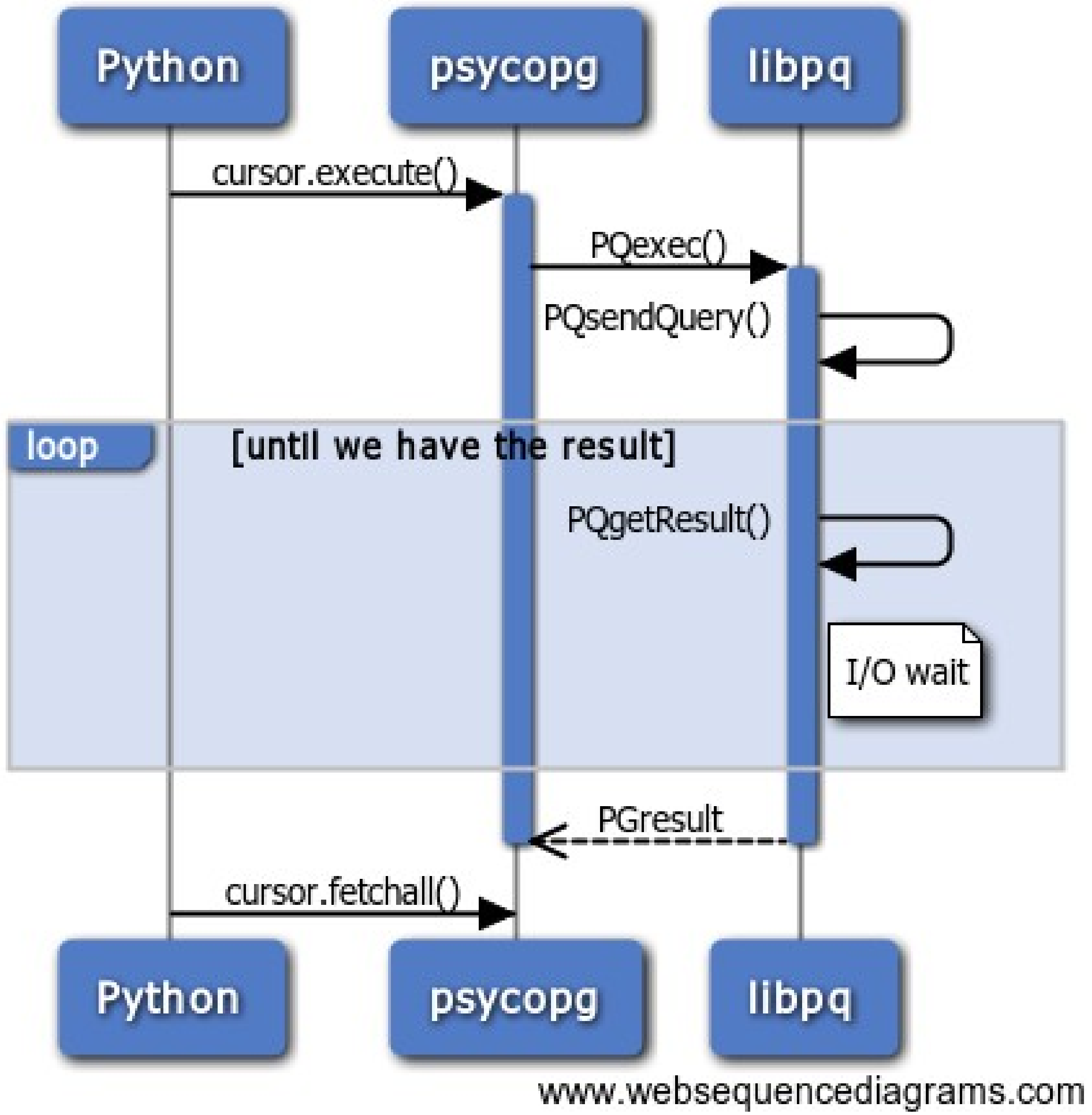
Async in psycopg

- Attempt from psycopg2, never worked correctly
  - `conn.execute(query, args, async=1)`
- Redesign in spring 2010, released in 2.2
  - Thanks to Jan Urbański
- Being async is now a connection property
  - `psycopg2.connect(dsn, async=1)`
  - Async code path well separated from sync

psycopg and libpq *sync*



psycopg and libpq *sync*



Async in psycopg

- `conn.fileno()`
  - Makes the connection a file-like object
- `conn.poll()` → [**OK**|**READ**|**WRITE**]

`poll()` knows things

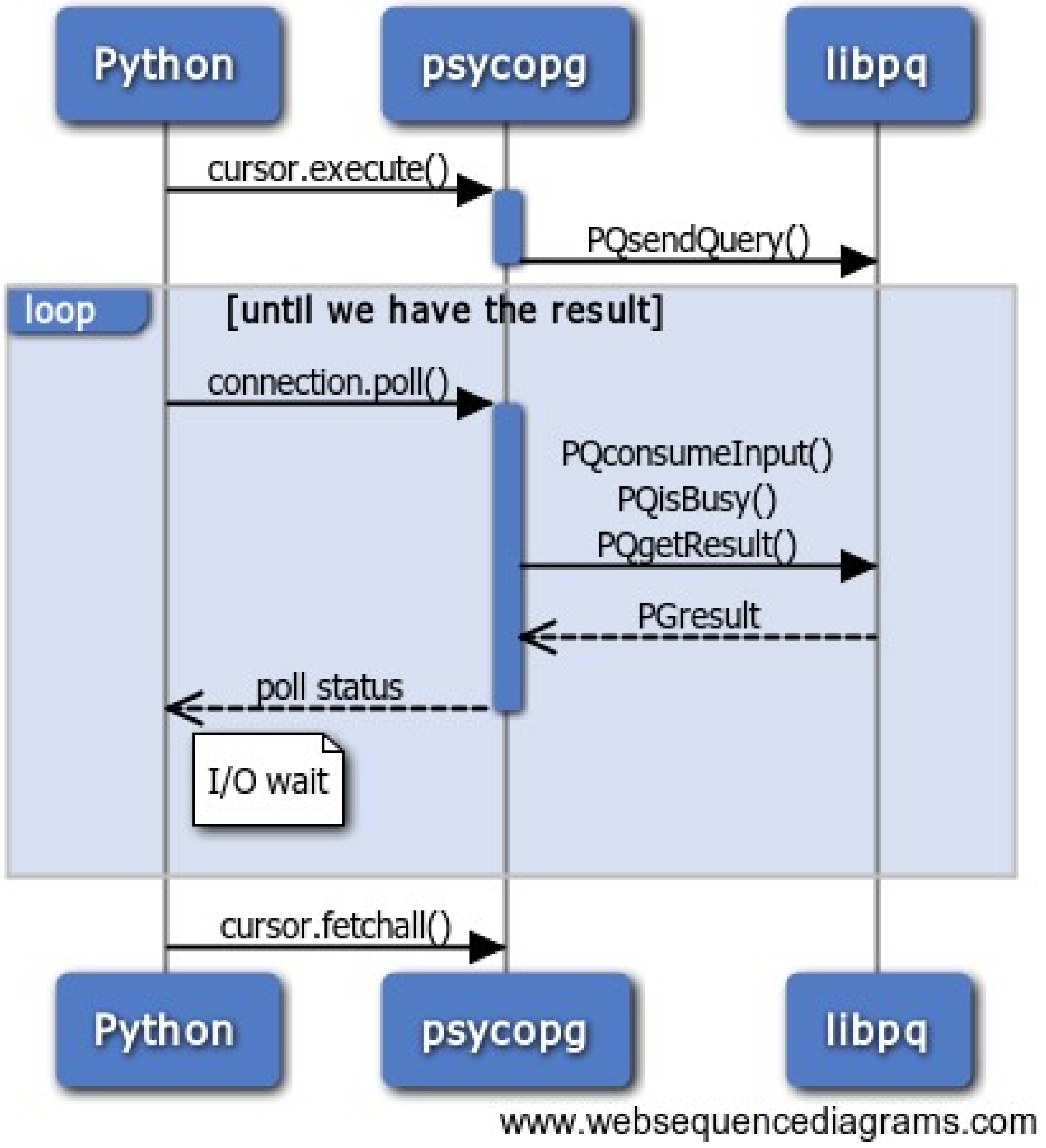
- Calls the correct libpq function
  - according to the operation to be performed
    - connection, query, fetch, notifies…
  - and the state of the connection
- Allows easy usage pattern

  -
    ```
    cur.execute(query, args)
    while “not_happy”:
        conn.poll()
    ```

Async example

```
cursor.execute(SQL)
while 1:
    state = conn.poll()
    if state == POLL_OK:
        break
    elif state == POLL_READ:
        select([conn.fileno()], [], [])
    elif state == POLL_WRITE:
        select([], [conn.fileno()], [])
cursor.fetchall()
```

psycopg and libpq *async*



Asynchronous access

- Fundamental problem: *DB-API is blocking*
  - `cnn = psycopg2.connect(dsn)`
  - `cursor.execute(query, args)`
  - `cursor.fetchall()`
- Async connections have a different interface
  - So we can't use Django, SQLAlchemy...
- Complete control, but higher level to be redone

Solution #1

- The “Twisted Solution”: what problem? :o)
  - everything must be callback-based anyway
- txPostgres: async psycopg2 in Twisted

```
d = conn.connect(database=DB_NAME)
d.addCallback(lambda c: c.execute(SQL))
d.addCallback(lambda c: c.fetchall())
```

- Notice: many features missing in async
  - No transactions, SSC, …

Coroutine libraries

- Interpreter-level cooperative aka “green” threads
  - Eventlet, gevent, uGreen
- “Monkeypatch” blocking functions
  - `time.sleep()`, `socket.read()`…
- C extensions can't be patched
  - A colleague of mine was struggling with pg8000…

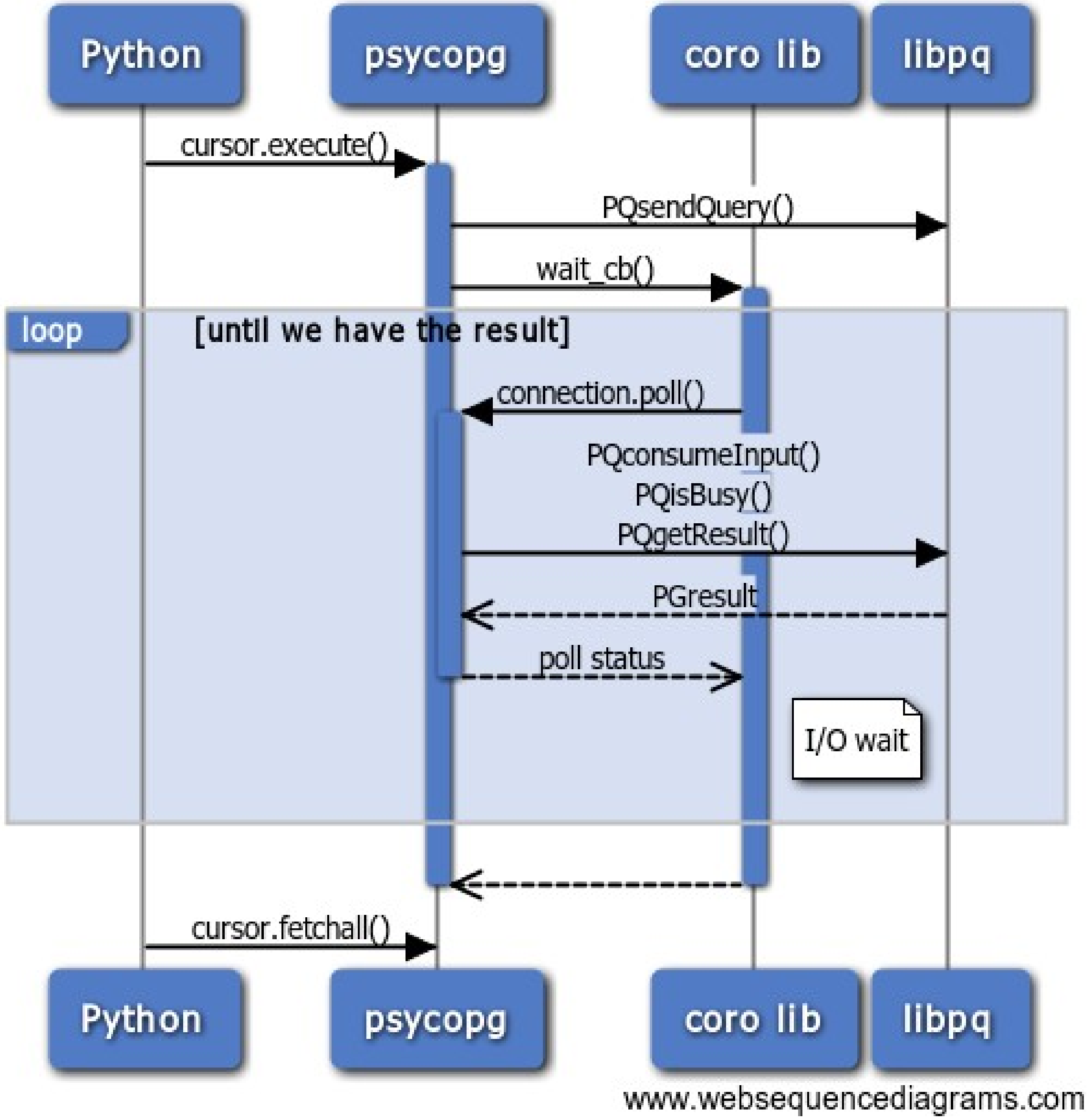
Solution #2: “wait” callback

- Globally registered
  - `psycopg2.extensions`
    `   .set_wait_callback(f)`
- Gives control back to the framework when it's time to wait
  - Control can be passed to a different thread
- The Python interface is unchanged
  - Less flexible, but classic blocking DB-API
- Customized for different coroutine libraries
  - Outside of psycopg scope, but check psycogreen

Example wait callback

```
def eventlet_wait_callback(conn):
  while 1:
    state = conn.poll()
    if state == POLL_OK:
      break
    elif state == POLL_READ:
      trampoline(conn.fileno(), read=1)
    elif state == POLL_WRITE:
      trampoline(conn.fileno(), write=1)
```

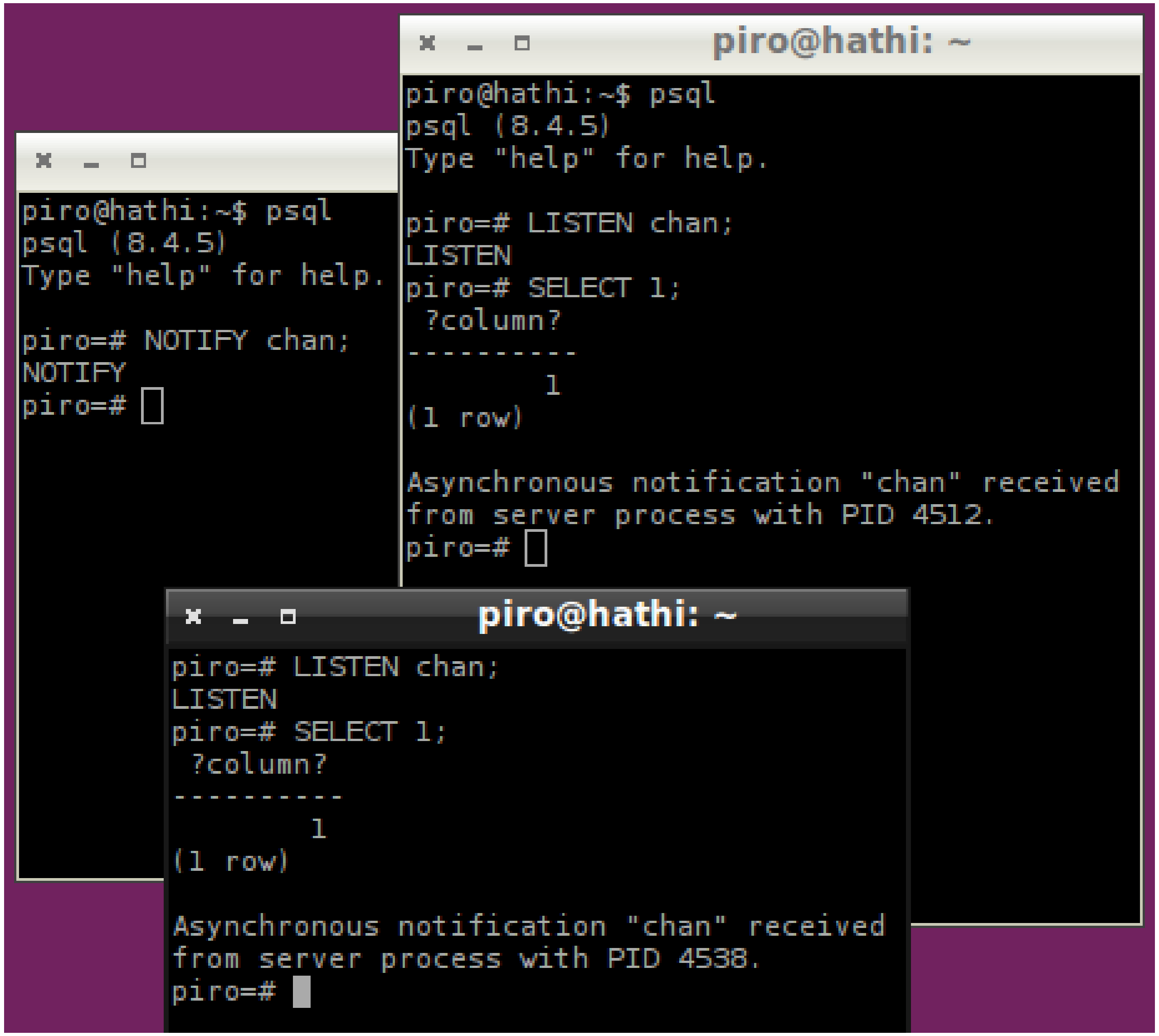
psycopg and libpq *green*



- Types adaptation
- Server-side cursors
- Transactions handling
- Async support
- **Server notifications**

Server notifications

- Publish/ subscribe channels
- PostgreSQL `LISTEN` and `NOTIFY`
- Added payload in PostgreSQL 9.0



```
piro@hathi:~$ psql
psql (8.4.5)
Type "help" for help.

piro=# NOTIFY chan;
NOTIFY
piro=#
```

```
piro@hathi:~$ psql
psql (8.4.5)
Type "help" for help.

piro=# LISTEN chan;
LISTEN
piro=# SELECT 1;
 ?column?
----------
        1
(1 row)

Asynchronous notification "chan" received
from server process with PID 4512.
piro=#
```

```
piro=# LISTEN chan;
LISTEN
piro=# SELECT 1;
 ?column?
----------
        1
(1 row)

Asynchronous notification "chan" received
from server process with PID 4538.
piro=#
```

Server notifications

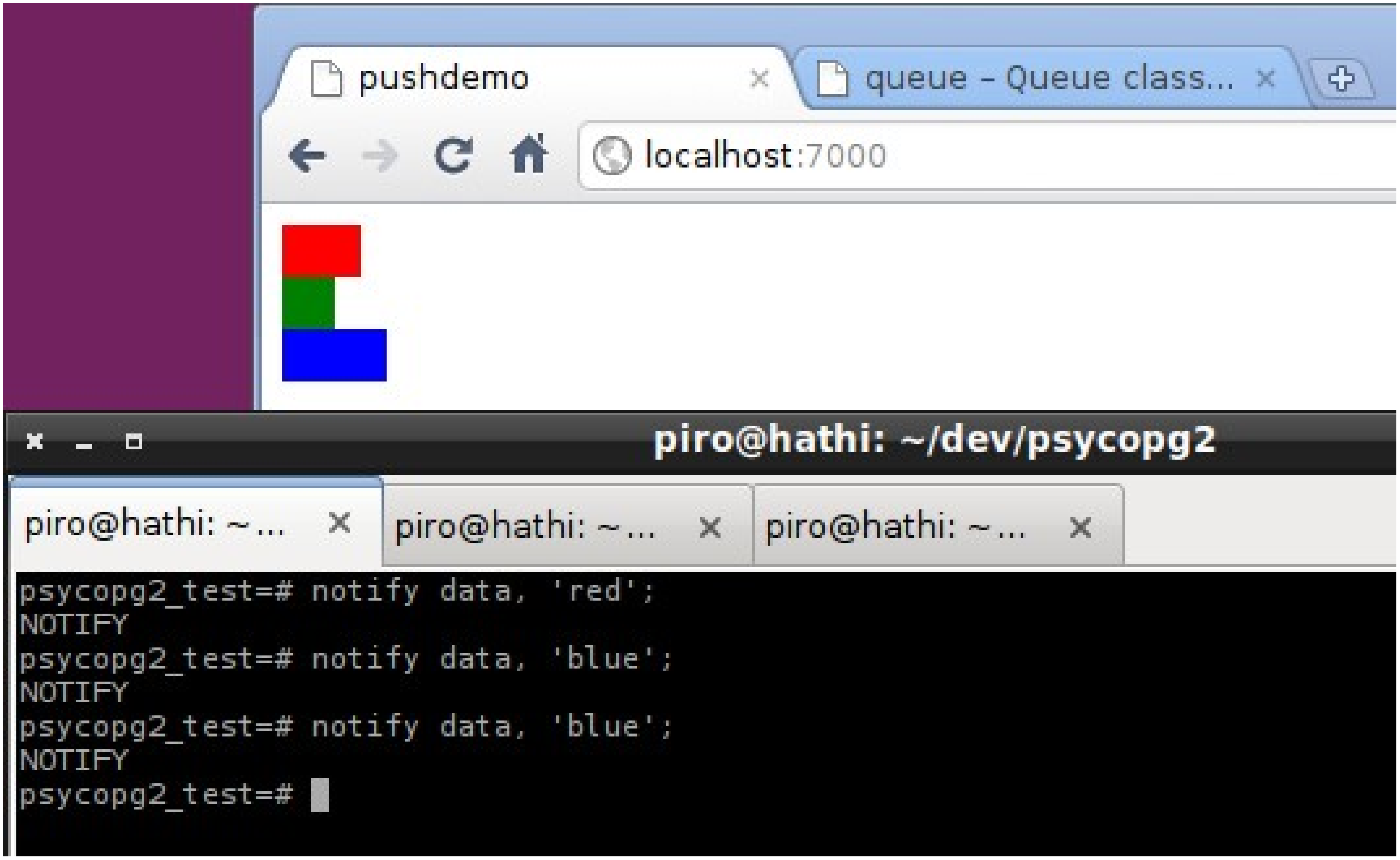
- Payload support from Psycopg 2.3
- Received on `execute()`
- Received on `poll()`
- They **love** async mode!

Notification: push example

- Listen for DB notifies and put them in a queue

```
def dblisten(q):
    cnn = psycopg2.connect(dsn)
    cnn.set_isolation_level(0)
    cur = cnn.cursor()
    cur.execute("listen data;")
    while 1:
        trampoline(cnn, read=True)
        cnn.poll()
        while cnn.notifies:
            q.put(cnn.notifies.pop())
```

Notification: push example



```
psycopg2_test=# notify data, 'red';
NOTIFY
psycopg2_test=# notify data, 'blue';
NOTIFY
psycopg2_test=# notify data, 'blue';
NOTIFY
psycopg2_test=#
```

Thanks!

Questions?