Unit Test Your Database!

David E. Wheeler
PostgreSQL Experts, Inc.

PGCon, May 21, 2009

PGX
POSTGRESQL
EXPERTS, INC.

Why?

Do these sound familiar?

“It takes too long to write tests.”

“Testing will just slow me down.”

“It takes too long to run tests.”

“We already write app-level unit tests.”

“I test stuff by running my app.”

"Tests never find

“This code is so simple it doesn’t need tests.”

“This function is too hard to test.”

“This is a private function.”

"Tests can't prove a program correct so why bother?"

"The behavior of the code changes a lot and rewriting the tests to match will just slow things down."

"If I imagined a problem to write tests for, I probably wrote code that doesn't have that problem."

"I can produce software that works even without focusing specifically on low-level unit tests."

“I’m lucky enough to only be dealing with really good developers.”

"AHHHHHHHH!!!! NOT TESTING! Anything but testing! Beat me, whip me, send me to Detroit, but don't make me write tests!"

—Michael Schwern, Test::Tutorial

Test Conceptions

Test Conceptions

- For finding bugs

Test Conceptions

- For finding bugs
- Difficult

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code
- Users test the code

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code
- Users test the code
- App tests are sufficient

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code
- Users test the code
- App tests are sufficient
- For public interface only

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code
- Users test the code
- App tests are sufficient
- For public interface only
- Prove nothing

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code
- Users test the code
- App tests are sufficient
- For public interface only
- Prove nothing
- For stable code

Test Conceptions

- For finding bugs
- Difficult
- Irrelevant
- Time-consuming
- For inexperienced developers
- Unnecessary for simple code
- Best for fragile code
- Users test the code
- App tests are sufficient
- For public interface only
- Prove nothing
- For stable code
- I really like Detroit

Let's Get Real

What does it take?

Test-Driven Development

Test-Driven Development

- Say you need a Fibonacci Calculator

Test-Driven Development

- Say you need a Fibonacci Calculator
- Start with a test

Test-Driven Development

- Say you need a Fibonacci Calculator
- Start with a test
- Write the simplest possible function

Test-Driven Development

- Say you need a Fibonacci Calculator
- Start with a test
- Write the simplest possible function
- Add more tests

Test-Driven Development

- Say you need a Fibonacci Calculator
- Start with a test
- Write the simplest possible function
- Add more tests
- Update the function

Test-Driven Development

- Say you need a Fibonacci Calculator
- Start with a test
- Write the simplest possible function
- Add more tests
- Update the function
- Wash, rinse, repeat...

Simple Test

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Simple Test

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT * FROM no_plan();

SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Simple Function

Simple Function

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN 0;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Simple Function

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN 0;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Run the Test

Terminal

%

Run the Test

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
%
```

Run the Test

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
1..2
ok
All tests successful.
Files=1, Tests=2, 0 secs (0.03 usr + 0.00 sys = 0.03 CPU)
Result: PASS
%
```

Run the Test

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
1..2
ok
All tests successful.
Files=1, Tests=2, 0 secs (0.03 usr + 0.00 sys = 0.03 CPU)
Result: PASS
%
```

Run the Test

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
1..2
ok
All tests successful.
Files=1, Tests=2, 0 secs (0.03 usr + 0.00 sys = 0.03 CPU)
Result: PASS
%
```

That was easy

Add Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
```

--:-- try.sql All (SQL[ansi])

Add Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
```

--:-- try.sql All (SQL[ansi])

Add Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
```

--:-- try.sql All (SQL[ansi])

Add Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
```

--:-- try.sql All (SQL[ansi])

Terminal

```
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
not ok 4 - fib(1) should be 1
# Failed test 4: "fib(1) should be 1"
#         have: 0
#         want: 1
1..4
# Looks like you failed 1 test of 4
Failed 1/4 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 4 Failed: 1)
  Failed test:  4
Files=1, Tests=4, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
not ok 4 - fib(1) should be 1
# Failed test 4: "fib(1) should be 1"
#         have: 0
#         want: 1
1..4
# Looks like you failed 1 test of 4
Failed 1/4 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 4 Failed: 1)
  Failed test:  4
Files=1, Tests=4, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Modify for the Test

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN 0;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Modify for the Test

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN fib_for;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Modify for the Test

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN fib_for;
END;
$$ LANGUAGE plpgsql;
```

Bare minimum

--:-- try.sql All (SQL[ansi])

Tests Pass!

Terminal

%

Tests Pass!

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
ok 4 - fib(1) should be 1
1..4
ok
All tests successful.
Files=1, Tests=4, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

Add Another Assertion

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
```

--:-- try.sql All (SQL[ansi])

Add Another Assertion

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
```

--:-- try.sql All (SQL[ansi])----------

Terminal

```
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
ok 4 - fib(1) should be 1
not ok 5 - fib(2) should be 1
# Failed test 5: "fib(2) should be 1"
#         have: 2
#         want: 1
1..5
# Looks like you failed 1 test of 5
Failed 1/5 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 5 Failed: 1)
  Failed test:  5
Files=1, Tests=5, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
ok 4 - fib(1) should be 1
not ok 5 - fib(2) should be 1
# Failed test 5: "fib(2) should be 1"
#         have: 2
#         want: 1
1..5
# Looks like you failed 1 test of 5
Failed 1/5 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 5 Failed: 1)
  Failed test:  5
Files=1, Tests=5, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Modify to Pass

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    RETURN fib_for;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Modify to Pass

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
    IF fib_for < 2 THEN
        RETURN fib_for;
    END IF;
    RETURN fib_for - 1;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

And...Pass!

Terminal

%

And...Pass!

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql ..
ok 1 - Schema pg_catalog or public or tap can
ok 2 - Function fib(integer) should exist
ok 3 - fib(0) should be 0
ok 4 - fib(1) should be 1
ok 5 - fib(2) should be 1
1..5
ok
All tests successful.
Files=1, Tests=5, 0 secs (0.02 usr + 0.00 sys = 0.02 CPU)
Result: PASS
%
```

Still More Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
```

--:-- try.sql All (SQL[ansi])

Still More Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
```

--:-- try.sql All (SQL[ansi])

Terminal

```
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sq1 .. 1/?
not ok 8 - fib(5) should be 5
# Failed test 8: "fib(5) should be 5"
#         have: 4
#         want: 5
# Looks like you failed 1 test of 8
test_fib.sql .. Failed 1/8 subtests

Test Summary Report
-------------------
test_fib.sq1 (Wstat: 0 Tests: 8 Failed: 1)
  Failed test:  8
Files=1, Tests=8, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 .. 1/?
not ok 8 - fib(5) should be 5
# Failed test 8: "fib(5) should be 5"
#          have: 4
#          want: 5
# Looks like you failed 1 test of 8
test_fib.sql .. Failed 1/8 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 8 Failed: 1)
  Failed test:  8
Files=1, Tests=8, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 .. 1/?
not ok 8 - fib(5) should be 5
# Failed test 8: "fib(5) should be 5"
#          have: 4
#          want: 5
# Looks like you failed 1 test of 8
test_fib.sql .. Failed 1/8 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 8 Failed: 1)
  Failed test:  8
Files=1, Tests=8, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Fix The Function

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
   IF fib_for < 2 THEN
        RETURN fib_for;
   END IF;
   RETURN fib_for - 1;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

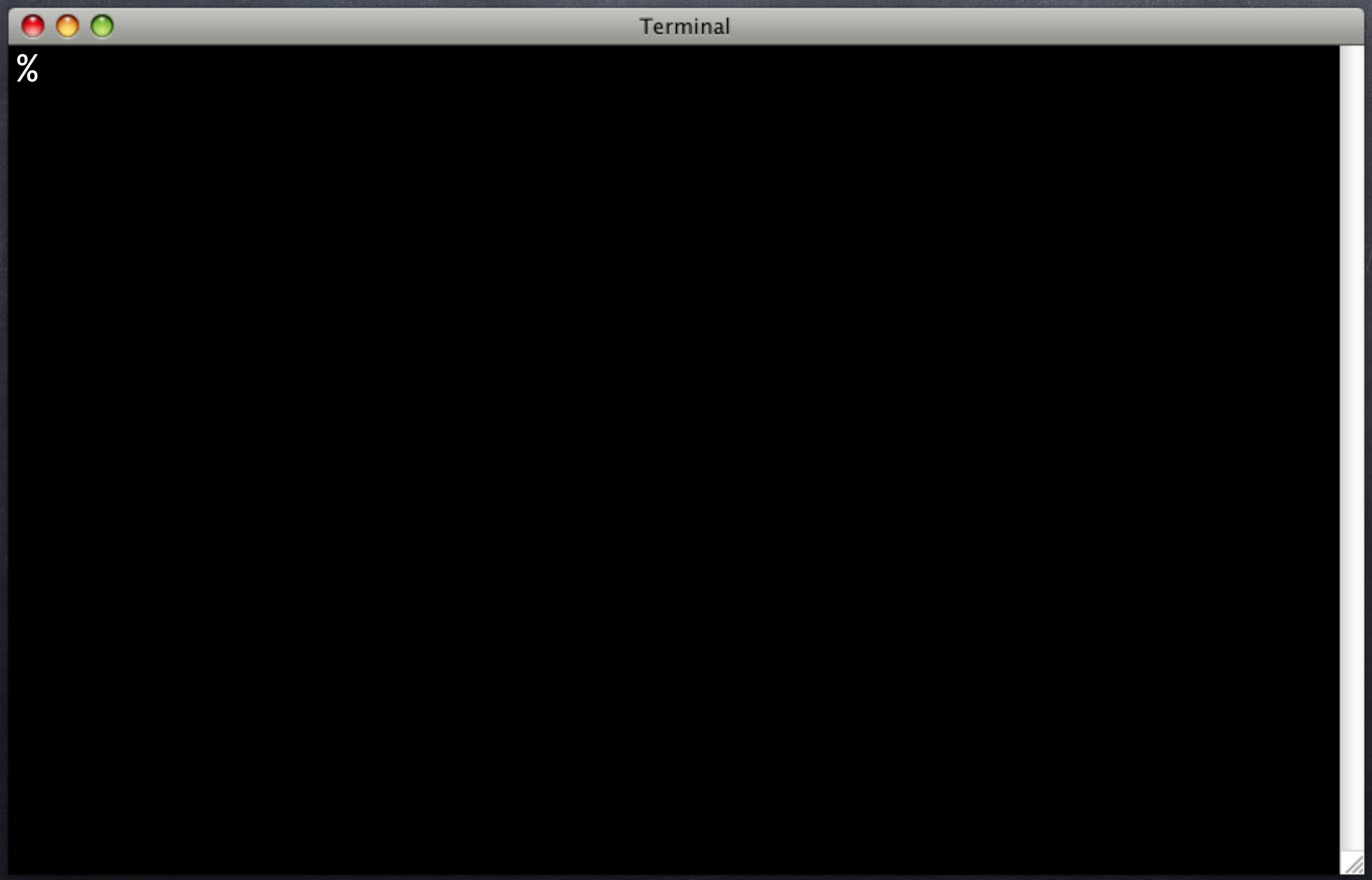
Fix The Function

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
   IF fib_for < 2 THEN
        RETURN fib_for;
   END IF;
   RETURN fib(fib_for - 2)
        + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

WOOT!



WOOT!

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. ok
All tests successful.
Files=1, Tests=8, 0 secs (0.02 usr + 0.00 sys = 0.02 CPU)
Result: PASS
%
```

A Few More Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
```

--:-- try.sql All (SQL[ansi])

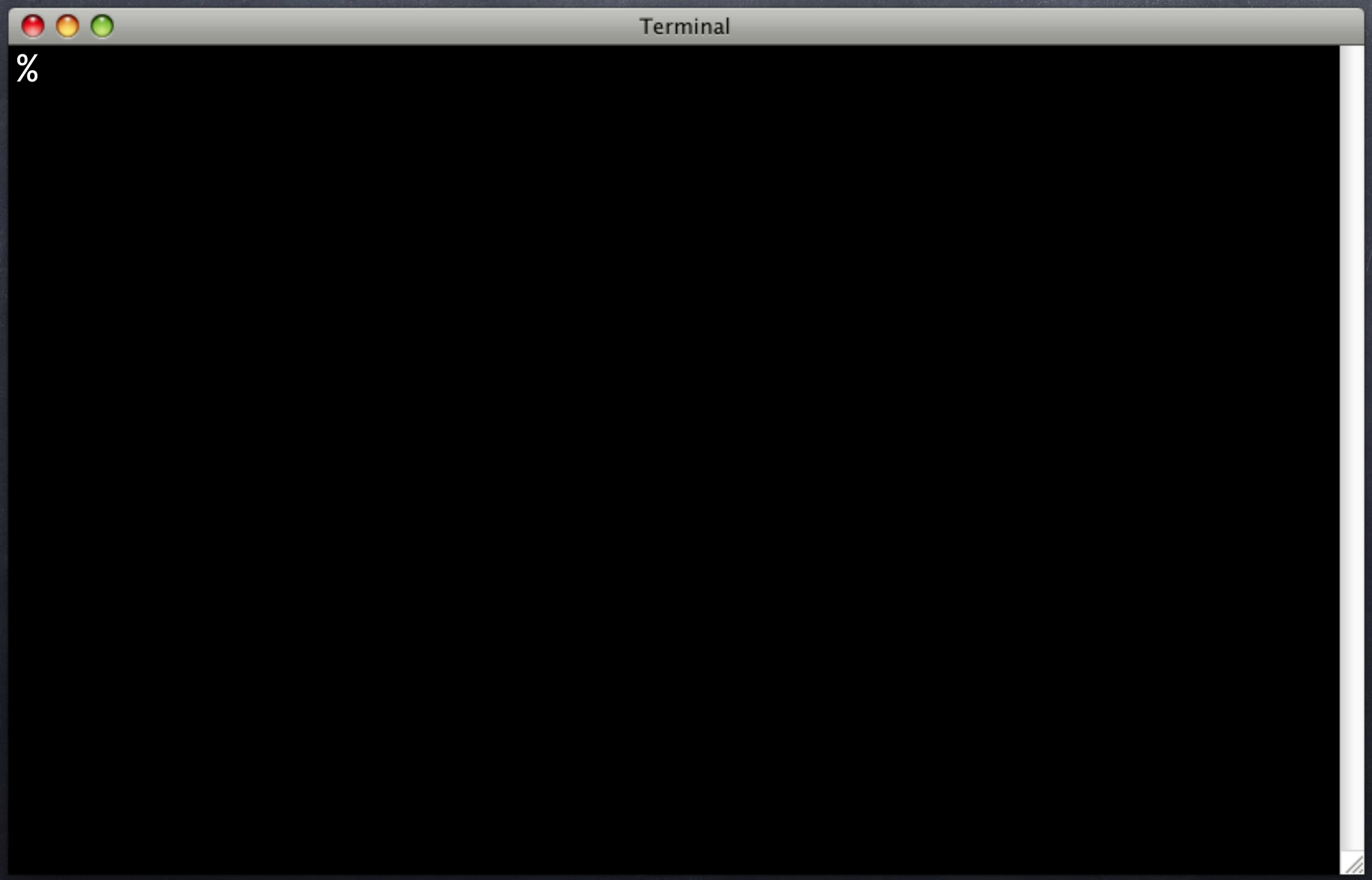
A Few More Assertions

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
```

--:-- try.sql All (SQL[ansi])

We're Golden!



We're Golden!

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. ok
All tests successful.
Files=1, Tests=11, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

Make it so,
Number One.

OMG

WTF???

The server is hammered!

Debug,
debug,
debug...

Nailed it!

Detroit, we have a problem.

Terminal

```
try=#
```

Detroit, we have a problem.

Terminal

```
try=# \timing
Timing is on.
try=# select fib(30);

  fib
--------
 832040
(1 row)

Time: 6752.112 ms
try=#
```

Detroit, we have a problem.

Terminal

```
try=# \timing
Timing is on.
try=# select fib(30);

  fib
--------
 832040
(1 row)

Time: 6752.112 ms
try=#
```

Regression

Add a Regression Test

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
```

--:-- try.sql All (SQL[ansi])

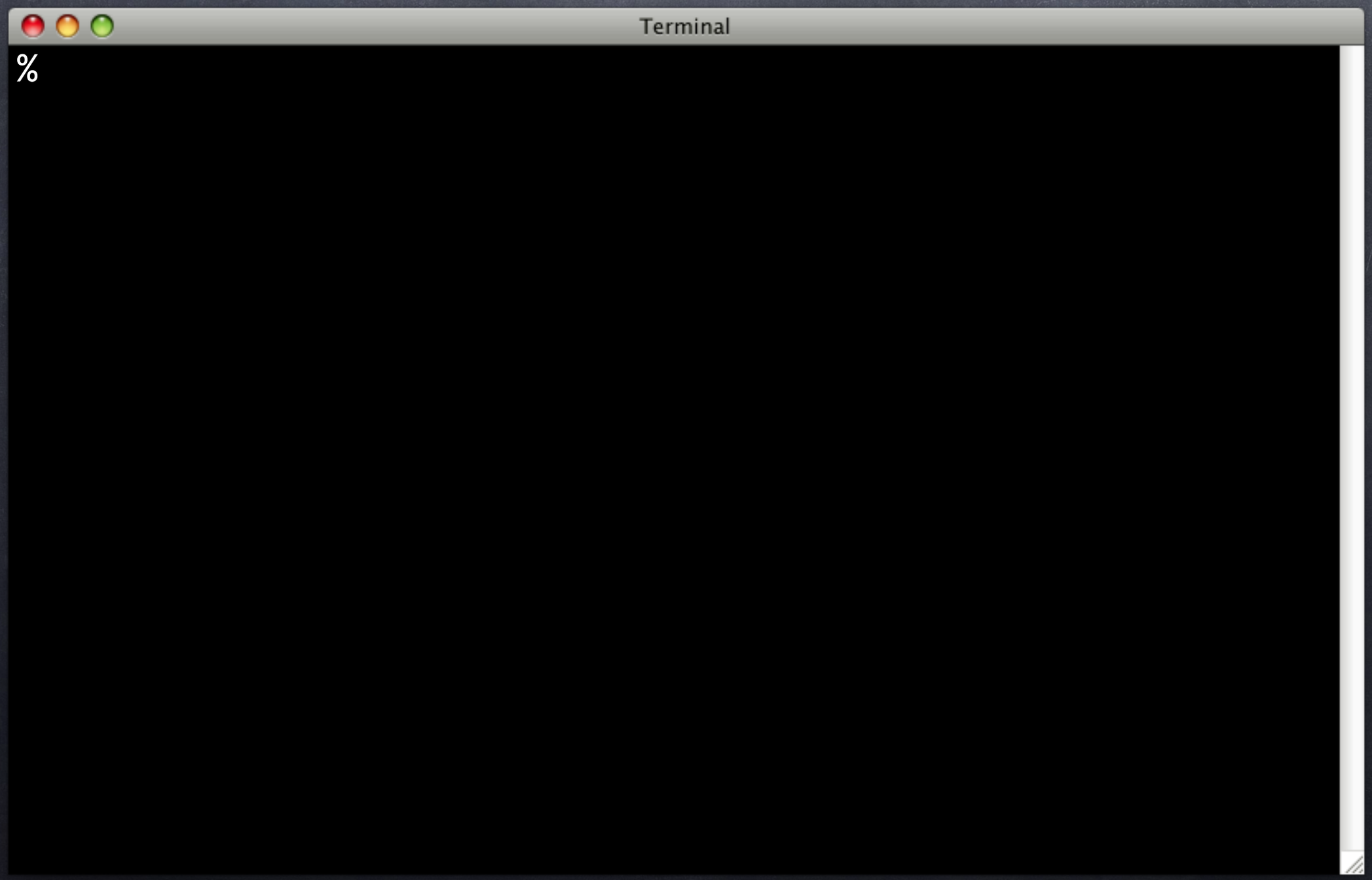
Add a Regression Test

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
SELECT performs_ok( 'SELECT fib(30)', 500 );
```

--:-- try.sql All (SQL[ansi])

What've We Got?



What've We Got?

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. 12/?
not ok 12 - Should run in less than 500 ms
# Failed test 12: "Should run in less than 500 ms"
#       runtime: 8418.816 ms
#       exceeds: 500 ms
# Looks like you failed 1 test of 12
test_fib.sql .. Failed 1/12 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 12 Failed: 1)
  Failed test:  12
Files=1, Tests=12, 8 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

What've We Got?

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. 12/?
not ok 12 - Should run in less than 500 ms
# Failed test 12: "Should run in less than 500 ms"
#       runtime: 8418.816 ms
#       exceeds: 500 ms
# Looks like you failed 1 test of 12
test_fib.sql .. Failed 1/12 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 12 Failed: 1)
  Failed test:  12
Files=1, Tests=12, 8 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL
%
```

Refactor

Emacs

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------
```

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
DECLARE
    ret integer := 0;
    nxt integer := 1;
    tmp integer;
BEGIN
    FOR num IN 0..fib_for LOOP
        tmp := ret;
        ret := nxt;
        nxt := tmp + nxt;
    END LOOP;

    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Terminal

```
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 .. 1/?
not ok 3 - fib(0) should be 0
# Failed test 3: "fib(0) should be 0"
#         have: 1
#         want: 0
not ok 5 - fib(2) should be 1
# Failed test 5: "fib(2) should be 1"
#         have: 2
#         want: 1
not ok 6 - fib(3) should be 2
# Failed test 6: "fib(3) should be 2"
#         have: 3
#         want: 2
not ok 7 - fib(4) should be 3
# Failed test 7: "fib(4) should be 3"
#         have: 5
#         want: 3
not ok 8 - fib(5) should be 5
# Failed test 8: "fib(5) should be 5"
#         have: 8
#         want: 5
not ok 9 - fib(6) Should be 8
# Failed test 9: "fib(6) Should be 8"
#         have: 13
#         want: 8
not ok 10 - fib(7) Should be 13
# Failed test 10: "fib(7) Should be 13"
#         have: 21
#         want: 13
not ok 11 - fib(8) Should be 21
# Failed test 11: "fib(8) Should be 21"
#         have: 34
#         want: 21
# Looks like you failed 8 tests of 12
test_fib.sql .. Failed 8/12 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 12 Failed: 8)
  Failed tests:  3, 5-11
Files=1, Tests=12,  0 secs (0.03 usr + 0.01 sys = 0.04 CPU)
Result: FAIL
%
```

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql1 .. 1/?
not ok 3 - fib(0) should be 0
# Failed test 3: "fib(0) should be 0"
#         have: 1
#         want: 0
not ok 5 - fib(2) should be 1
# Failed test 5: "fib(2) should be 1"
#         have: 2
#         want: 1
not ok 6 - fib(3) should be 2
# Failed test 6: "fib(3) should be 2"
#         have: 3
#         want: 2
not ok 7 - fib(4) should be 3
# Failed test 7: "fib(4) should be 3"
#         have: 5
#         want: 3
not ok 8 - fib(5) should be 5
# Failed test 8: "fib(5) should be 5"
#         have: 8
#         want: 5
not ok 9 - fib(6) Should be 8
# Failed test 9: "fib(6) Should be 8"
#         have: 13
#         want: 8
not ok 10 - fib(7) Should be 13
# Failed test 10: "fib(7) Should be 13"
#         have: 21
#         want: 13
not ok 11 - fib(8) Should be 21
# Failed test 11: "fib(8) Should be 21"
#         have: 34
#         want: 21
# Looks like you failed 8 tests of 12
test_fib.sql .. Failed 8/12 subtests

Test Summary Report
-------------------
test_fib.sql (Wstat: 0 Tests: 12 Failed: 8)
  Failed tests:  3, 5-11
Files=1, Tests=12,  0 secs (0.03 usr + 0.01 sys = 0.04 CPU)
Result: FAIL
%
```

WTF?

Emacs

```sql
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
DECLARE
    ret integer := 0;
    nxt integer := 1;
    tmp integer;
BEGIN
    FOR num IN 0..fib_for LOOP
        tmp := ret;
        ret := nxt;
        nxt := tmp + nxt;
    END LOOP;

    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

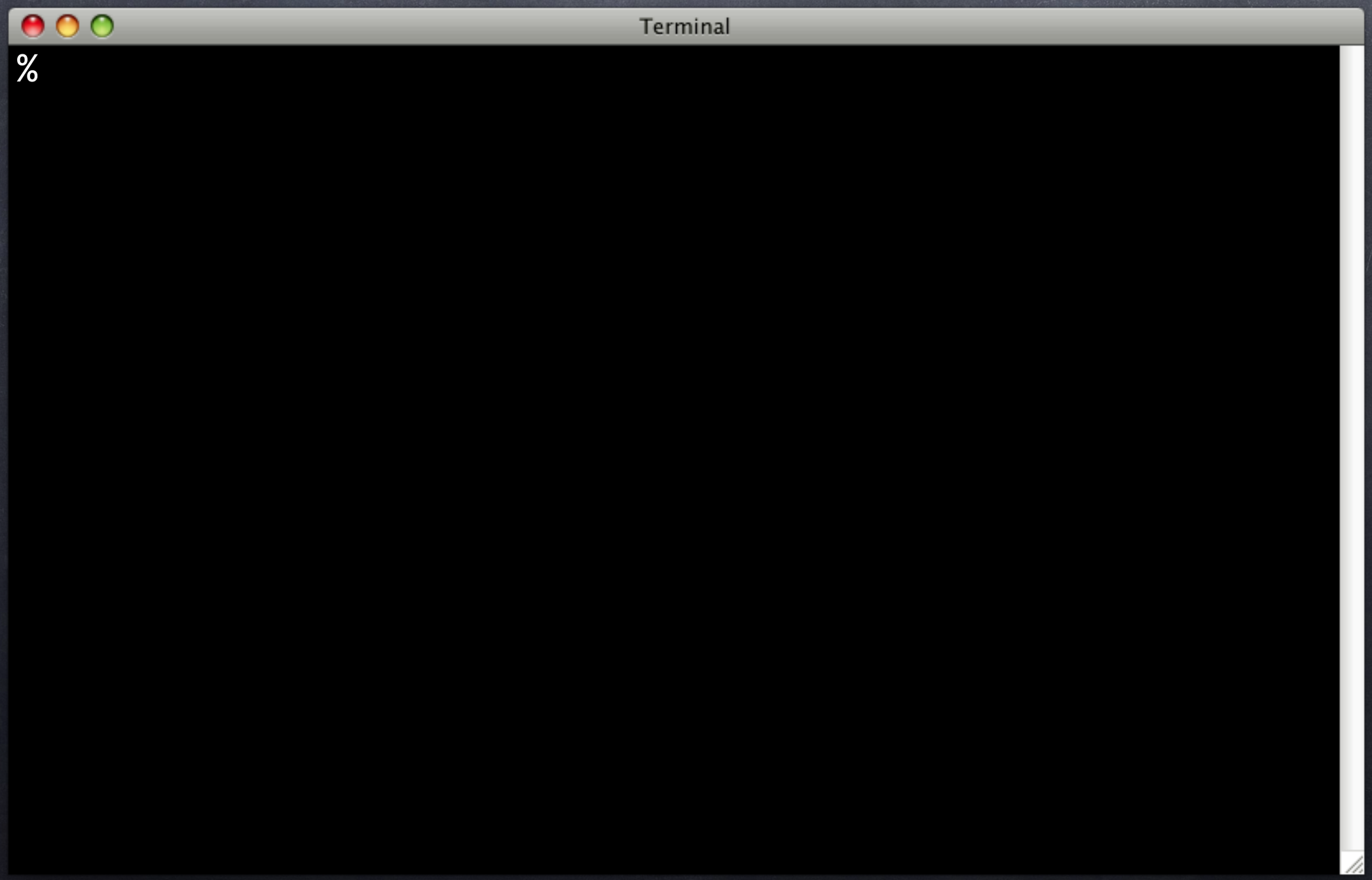
--:-- try.sql All (SQL[ansi])

Emacs

```
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
DECLARE
    ret integer := 0;
    nxt integer := 1;
    tmp integer;
BEGIN
    FOR num IN 1..fib_for LOOP
        tmp := ret;
        ret := nxt;
        nxt := tmp + nxt;
    END LOOP;

    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Back in Business



Back in Business

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. ok
All tests successful.
Files=1, Tests=12, 1 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

Just for the Hell of it...

Push It!

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
SELECT performs_ok( 'SELECT fib(30)', 500 );
```

--:-- try.sql All (SQL[ansi])

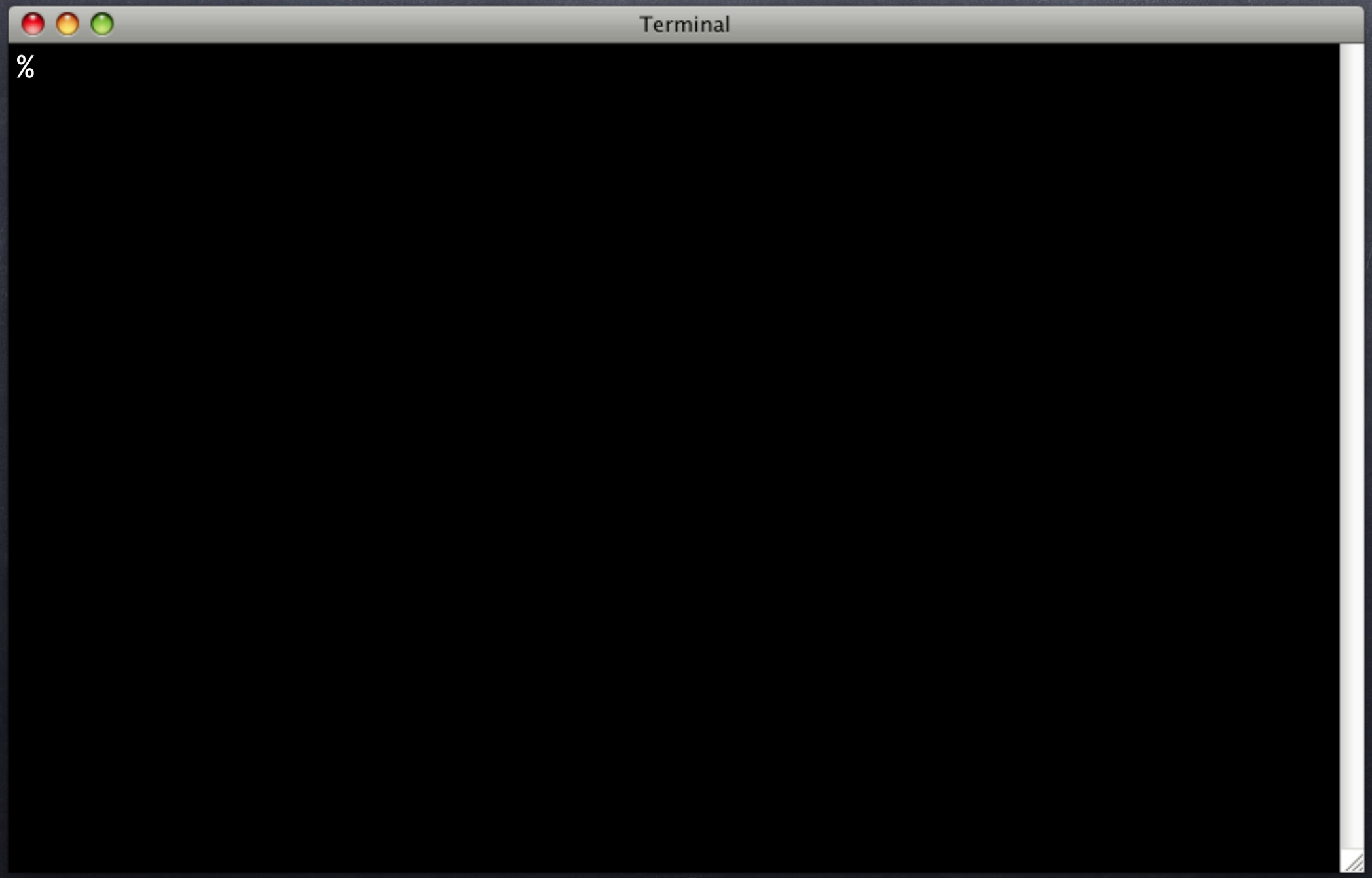
Push It!

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
SELECT performs_ok( 'SELECT fib(30)', 500 );
SELECT is( fib(32), 2178309, 'fib(32) is 2178309');
SELECT is( fib(64), 10610209857723, 'fib(64) is 10610209857723');
```

--:-- try.sql All (SQL[ansi])

No Fibbing.



No Fibbing.

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. 1/? psql:test_fib.sql:18: ERROR:  function is(integer,
bigint, unknown) does not exist
LINE 1: SELECT is( fib(64), 10610209857723, 'fib(64) Should be 10610...
               ^
HINT:  No function matches the given name and argument types. You might
need to add explicit type casts.
test_fib.sql .. Dubious, test returned 3 (wstat 768, 0x300)
All 13 subtests passed

Test Summary Report
-------------------
test_fib.sql (Wstat: 768 Tests: 13 Failed: 0)
  Non-zero exit status: 3
  Parse errors: No plan found in TAP output
Files=1, Tests=13, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL

%
```

No Fibbing.

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. 1/? psql:test_fib.sql:18: ERROR:  function is(integer,
bigint, unknown) does not exist
LINE 1: SELECT is( fib(64), 106102098577 23, 'fib(64) Should be 10610...
               ^
HINT:  No function matches the given name and argument types. You might
need to add explicit type casts.
test_fib.sql .. Dubious, test returned 3 (wstat 768, 0x300)
All 13 subtests passed

Test Summary Report
-------------------
test_fib.sql (Wstat: 768 Tests: 13 Failed: 0)
  Non-zero exit status: 3
  Parse errors: No plan found in TAP output
Files=1, Tests=13, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: FAIL

%
```

Hrm...

Emacs

```sql
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS integer AS $$
BEGIN
DECLARE
    ret integer := 0;
    nxt integer := 1;
    tmp integer;
BEGIN
    FOR num IN 1..fib_for LOOP
        tmp := ret;
        ret := nxt;
        nxt := tmp + nxt;
    END LOOP;

    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
CREATE OR REPLACE FUNCTION fib (
    fib_for integer
) RETURNS bigint  AS $$
BEGIN
DECLARE
    ret bigint  := 0;
    nxt bigint  := 1;
    tmp bigint ;
BEGIN
    FOR num IN 1..fib_for LOOP
        tmp := ret;
        ret := nxt;
        nxt := tmp + nxt;
    END LOOP;

    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Apples to Apples...

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0, 'fib(0) should be 0');
SELECT is( fib(1), 1, 'fib(1) should be 1');
SELECT is( fib(2), 1, 'fib(2) should be 1');
SELECT is( fib(3), 2, 'fib(3) should be 2');
SELECT is( fib(4), 3, 'fib(4) should be 3');
SELECT is( fib(5), 5, 'fib(5) should be 5');
SELECT is( fib(6), 8, 'fib(6) should be 8');
SELECT is( fib(7), 13, 'fib(7) should be 13');
SELECT is( fib(8), 21, 'fib(8) should be 21');
SELECT performs_ok( 'SELECT fib(30)', 500 );
SELECT is( fib(32), 2178309, 'fib(32) is 2178309');
SELECT is( fib(64), 10610209857723, 'fib(64) is 10610209857723');
```

--:-- try.sql All (SQL[ansi])

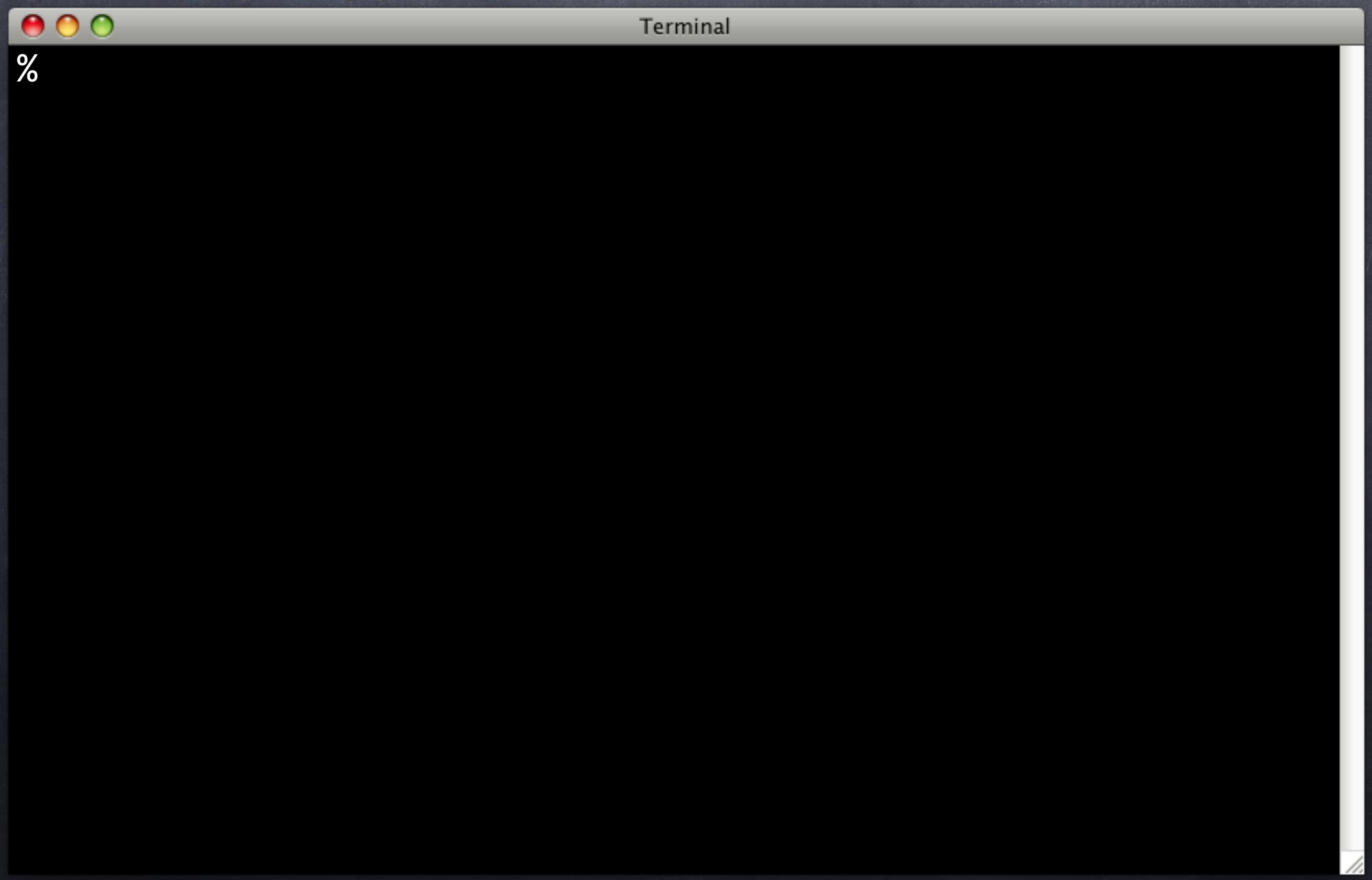
Apples to Apples...

Emacs

```
SELECT can('{fib}');
SELECT can_ok('fib', ARRAY['integer']);
SELECT is( fib(0), 0::int8, 'fib(0) should be 0');
SELECT is( fib(1), 1::int8, 'fib(1) should be 1');
SELECT is( fib(2), 1::int8, 'fib(2) should be 1');
SELECT is( fib(3), 2::int8, 'fib(3) should be 2');
SELECT is( fib(4), 3::int8, 'fib(4) should be 3');
SELECT is( fib(5), 5::int8, 'fib(5) should be 5');
SELECT is( fib(6), 8::int8, 'fib(6) should be 8');
SELECT is( fib(7), 13::int8, 'fib(7) should be 13');
SELECT is( fib(8), 21::int8, 'fib(8) should be 21');
SELECT performs_ok( 'SELECT fib(30)', 500 );
SELECT is( fib(32), 2178309::int8, 'fib(32) is 2178309');
SELECT is( fib(64), 10610209857723::int8, 'fib(64) is 10610209857723');
```

--:-- try.sql All (SQL[ansi])

And Now?



And Now?

Terminal

```
% psql -d try -f fib.sql
CREATE FUNCTION
% pg_prove -vd try test_fib.sql
test_fib.sql .. ok
All tests successful.
Files=1, Tests=14, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

TDD Means Consistency

What about Maintenance?

Emacs

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------
```

Emacs

```
CREATE FUNCTION find_by_birthday(integer, integer, integer, integer, text)
RETURNS SETOF integer AS $$
DECLARE
    p_day    ALIAS FOR $1;
    p_mon    ALIAS FOR $2;
    p_offset ALIAS FOR $3;
    p_limit  ALIAS FOR $4;
    p_state  ALIAS FOR $5;
    v_qry    TEXT;
    v_output RECORD;
BEGIN
    v_qry := 'SELECT * FROM users WHERE state = ''' || p_state || '''';
    v_qry := v_qry || ' AND birth_mon ~ ''^0?' || p_mon::text || '$''';
    v_qry := v_qry || ' AND birth_day = ''' || p_day::text || '''';
    v_qry := v_qry || ' ORDER BY user_id';
    IF p_offset IS NOT NULL THEN
        v_qry := v_qry || ' OFFSET ' || p_offset;
    END IF;
    IF p_limit IS NOT NULL THEN
        v_qry := v_qry || ' LIMIT ' || p_limit;
    END IF;
    FOR v_output IN EXECUTE v_qry LOOP
        RETURN NEXT v_output.user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Emacs

```
CREATE FUNCTION find_by_birthday(integer, integer, integer, integer, text)
RETURNS SETOF integer AS $$
DECLARE
    p_day    ALIAS FOR $1;
    p_mon    ALIAS FOR $2;
    p_offset ALIAS FOR $3;
    p_limit  ALIAS FOR $4;
    p_state  ALIAS FOR $5;
    v_qry    TEXT;
    v_output RECORD;
BEGIN
    v_qry := 'SELECT * FROM users WHERE state = ''' || p_state || '''';
    v_qry := v_qry || ' AND birth_mon ~ ''^0?' || p_mon::text || '$''';
    v_qry := v_qry || ' AND birth_day = ''' || p_day::text || '''';
    v_qry := v_qry || ' ORDER BY user_id';
    IF p_offset IS NOT NULL THEN
        v_qry := v_qry || ' OFFSET ' || p_offset;
    END IF;
    IF p_limit IS NOT NULL THEN
        v_qry := v_qry || ' LIMIT ' || p_limit;
    END IF;
    FOR v_output IN EXECUTE v_qry LOOP
        RETURN NEXT v_output.user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Emacs

```
CREATE FUNCTION find_by_birthday(integer, integer, integer, integer, text)
RETURNS SETOF integer AS $$
DECLARE
    p_day    ALIAS FOR $1;
    p_mon    ALIAS FOR $2;
    p_offset ALIAS FOR $3;
    p_limit  ALIAS FOR $4;
    p_state  ALIAS FOR $5;
    v_qry    TEXT;
    v_output RECORD;
BEGIN
    v_qry := 'SELECT * FROM users WHERE state = ''' || p_state || '''';
    v_qry := v_qry || ' AND birth_mon ~ ''^0?' || p_mon::text || '$''';
    v_qry := v_qry || ' AND birth_day = ''' || p_day::text || '''';
    v_qry := v_qry || ' ORDER BY user_id';
    IF p_offset IS NOT NULL THEN
        v_qry := v_qry || ' OFFSET ' || p_offset;
    END IF;
    IF p_limit IS NOT NULL THEN
        v_qry := v_qry || ' LIMIT ' || p_limit;
    END IF;
    FOR v_output IN EXECUTE v_qry LOOP
        RETURN NEXT v_output.user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Emacs

```
CREATE FUNCTION find_by_birthday(integer, integer, integer, integer, text)
RETURNS SETOF integer AS $$
DECLARE
    p_day    ALIAS FOR $1;
    p_mon    ALIAS FOR $2;
    p_offset ALIAS FOR $3;
    p_limit  ALIAS FOR $4;
    p_state  ALIAS FOR $5;
    v_qry    TEXT;
    v_output RECORD;
BEGIN
    v_qry := 'SELECT * FROM users WHERE state = ''' || p_state || '''';
    v_qry := v_qry || ' AND birth_mon ~ ''^0?' || p_mon::text || '$''';
    v_qry := v_qry || ' AND birth_day = ''' || p_day::text || '''';
    v_qry := v_qry || ' ORDER BY user_id';
    IF p_offset IS NOT NULL THEN
        v_qry := v_qry || ' OFFSET ' || p_offset;
    END IF;
    IF p_limit IS NOT NULL THEN
        v_qry := v_qry || ' LIMIT ' || p_limit;
    END IF;
    FOR v_output IN EXECUTE v_qry LOOP
        RETURN NEXT v_output.user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Emacs

```
CREATE FUNCTION find_by_birthday(integer, integer, integer, integer, text)
RETURNS SETOF integer AS $$
DECLARE
    p_day    ALIAS FOR $1;
    p_mon    ALIAS FOR $2;
    p_offset ALIAS FOR $3;
    p_limit  ALIAS FOR $4;
    p_state  ALIAS FOR $5;
    v_qry    TEXT;
    v_output RECORD;
BEGIN
    v_qry := 'SELECT * FROM users WHERE state = ''' || p_state || '''';
    v_qry := v_qry || ' AND birth_mon ~ ''^0?' || p_mon::text || '$''';
    v_qry := v_qry || ' AND birth_day = ''' || p_day::text || '''';
    v_qry := v_qry || ' ORDER BY user_id';
    IF p_offset IS NOT NULL THEN
        v_qry := v_qry || ' OFFSET ' || p_offset;
    END IF;
    IF p_limit IS NOT NULL THEN
        v_qry := v_qry || ' LIMIT ' || p_limit;
    END IF;
    FOR v_output IN EXECUTE v_qry LOOP
        RETURN NEXT v_output.user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

Emacs

```
CREATE FUNCTION find_by_birthday(integer, integer, integer, integer, text)
RETURNS SETOF integer AS $$
DECLARE
    p_day    ALIAS FOR $1;
    p_mon    ALIAS FOR $2;
    p_offset ALIAS FOR $3;
    p_limit  ALIAS FOR $4;
    p_state  ALIAS FOR $5;
    v_qry    TEXT;
    v_output RECORD;
BEGIN
    v_qry := 'SELECT * FROM users WHERE state = ''' || p_state || '''';
    v_qry := v_qry || ' AND birth_mon ~ ''^0?' || p_mon::text || '$''';
    v_qry := v_qry || ' AND birth_day = ''' || p_day::text || '''';
    v_qry := v_qry || ' ORDER BY user_id';
    IF p_offset IS NOT NULL THEN
        v_qry := v_qry || ' OFFSET ' || p_offset;
    END IF;
    IF p_limit IS NOT NULL THEN
        v_qry := v_qry || ' LIMIT ' || p_limit;
    END IF;
    FOR v_output IN EXECUTE v_qry LOOP
        RETURN NEXT v_output.user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;
```

--:-- try.sql All (SQL[ansi])

What's on the Table?

Terminal

```
try=#
```

What's on the Table?

Terminal

```
try=# \d users
                          Table "public.users"
   Column   |         Type         |         Modifiers
------------+----------------------+----------------------------------------
 user_id    | integer              | not null default nextval(…)
 name       | text                 | not null default ''::text
 birthdate  | date                 |
 birth_mon  | character varying(2) |
 birth_day  | character varying(2) |
 birth_year | character varying(4) |
 state      | text                 | not null default 'active'::text
Indexes:
    "users_pkey" PRIMARY KEY, btree (user_id)
```

What's on the Table?

Terminal

```
try=# \d users
                          Table "public.users"
   Column   |          Type          |          Modifiers
------------+------------------------+-------------------------------------
 user_id    | integer                | not null default nextval(…)
 name       | text                   | not null default ''::text
 birthdate  | date                   |
 birth_mon  | character varying(2)   |
 birth_day  | character varying(2)   |
 birth_year | character varying(4)   |
 state      | text                   | not null default 'active'::text
Indexes:
    "users_pkey" PRIMARY KEY, btree (user_id)
```

What's on the Table?

Terminal

```
try=#
```

What's on the Table?

Terminal

```
try=# select * from users;
 user_id | name  | birthdate  | birth_mon | birth_day | birth_year | state
---------+-------+------------+-----------+-----------+------------+--------
       1 | David | 1968-12-19 | 12        | 19        | 1968       | active
       2 | Josh  | 1970-03-12 | 03        | 12        | 1970       | active
       3 | Dan   | 1972-06-03 | 6         | 3         | 1972       | active
(3 rows)
```

What's on the Table?

Terminal

```
try=# select * from users;
 user_id | name  | birthdate  | birth_mon | birth_day | birth_year | state
---------+-------+------------+-----------+-----------+------------+--------
       1 | David | 1968-12-19 | 12        | 19        | 1968       | active
       2 | Josh  | 1970-03-12 | 03        | 12        | 1970       | active
       3 | Dan   | 1972-06-03 | 6         | 3         | 1972       | active
(3 rows)
```

Must...
restrain...
fist...
of death...

The Situation

The Situation

- This is production code

The Situation

- This is production code
- Cannot afford downtime

The Situation

- This is production code
- Cannot afford downtime
- No room for mistakes

The Situation

- This is production code
- Cannot afford downtime
- No room for mistakes
- Bugs must remain consistent

The Situation

- This is production code
- Cannot afford downtime
- No room for mistakes
- Bugs must remain consistent
- But...

Dear GOD it needs rewriting.

But first...

Test the existing implementation.

Emacs

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------
```

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_is_pk(    'users', 'user_id' );
SELECT col_not_null( 'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_is_pk(    'users', 'user_id' );
SELECT col_not_null( 'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_is_pk(    'users', 'user_id' );
SELECT col_not_null( 'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_is_pk(    'users', 'user_id' );
SELECT col_not_null( 'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_is_pk(    'users', 'user_id' );
SELECT col_not_null( 'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Schema Sanity

Terminal

%

Schema Sanity

Terminal

```
% pg_prove -d try test_schema.sql
test_schema.sql .. ok
All tests successful.
Files=1, Tests=13,  0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

Emacs

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------
```

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
--SELECT plan(15);
SELECT * FROM no_plan();

SELECT can('{find_by_birthday}');
SELECT can_ok(
    'find_by_birthday',
    ARRAY['integer', 'integer', 'integer', 'integer', 'text']
);

-- Set up fixtures.
ALTER SEQUENCE users_user_id_seq RESTART 1;
INSERT INTO users (name, birthdate, birth_mon, birth_day, birth_year)
VALUES ('David', '1968-12-19', '12', '19', '1968'),
       ('Josh',  '1970-03-12', '03', '12', '1970'),
       ('Dan',   '1972-06-03', '6',  '3',  '1972'),
       ('Anna',  '2005-06-03', '06', '3',  '2005');

SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
--SELECT plan(15);
SELECT * FROM no_plan();

SELECT can('{find_by_birthday}');
SELECT can_ok(
    'find_by_birthday',
    ARRAY['integer', 'integer', 'integer', 'integer', 'text']
);

-- Set up fixtures.
ALTER SEQUENCE users_user_id_seq RESTART 1;
INSERT INTO users (name, birthdate, birth_mon, birth_day, birth_year)
VALUES ('David', '1968-12-19', '12', '19', '1968'),
       ('Josh',  '1970-03-12', '03', '12', '1970'),
       ('Dan',   '1972-06-03', '6',  '3',  '1972'),
       ('Anna',  '2005-06-03', '06', '3',  '2005');

SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
--SELECT plan(15);
SELECT * FROM no_plan();

SELECT can('{find_by_birthday}');
SELECT can_ok(
    'find_by_birthday',
    ARRAY['integer', 'integer', 'integer', 'integer', 'text']
);

-- Set up fixtures.
ALTER SEQUENCE users_user_id_seq RESTART 1;
INSERT INTO users (name, birthdate, birth_mon, birth_day, birth_year)
VALUES ('David', '1968-12-19', '12', '19', '1968'),
       ('Josh',  '1970-03-12', '03', '12', '1970'),
       ('Dan',   '1972-06-03', '6',  '3',  '1972'),
       ('Anna',  '2005-06-03', '06', '3',  '2005');

SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
--SELECT plan(15);
SELECT * FROM no_plan();

SELECT can('{find_by_birthday}');
SELECT can_ok(
    'find_by_birthday',
    ARRAY['integer', 'integer', 'integer', 'integer', 'text']
);

-- Set up fixtures.
ALTER SEQUENCE users_user_id_seq RESTART 1;
INSERT INTO users (name, birthdate, birth_mon, birth_day, birth_year)
VALUES ('David', '1968-12-19', '12', '19', '1968'),
       ('Josh',  '1970-03-12', '03', '12', '1970'),
       ('Dan',   '1972-06-03', '6',  '3',  '1972'),
       ('Anna',  '2005-06-03', '06', '3',  '2005');

SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
--SELECT plan(15);
SELECT * FROM no_plan();

SELECT can('{find_by_birthday}');
SELECT can_ok(
    'find_by_birthday',
    ARRAY['integer', 'integer', 'integer', 'integer', 'text']
);

-- Set up fixtures.
ALTER SEQUENCE users_user_id_seq RESTART 1;
INSERT INTO users (name, birthdate, birth_mon, birth_day, birth_year)
VALUES ('David', '1968-12-19', '12', '19', '1968'),
       ('Josh',  '1970-03-12', '03', '12', '1970'),
       ('Dan',   '1972-06-03', '6',  '3',  '1972'),
       ('Anna',  '2005-06-03', '06', '3',  '2005');

SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);

SELECT * FROM finish();
ROLLBACK;
```

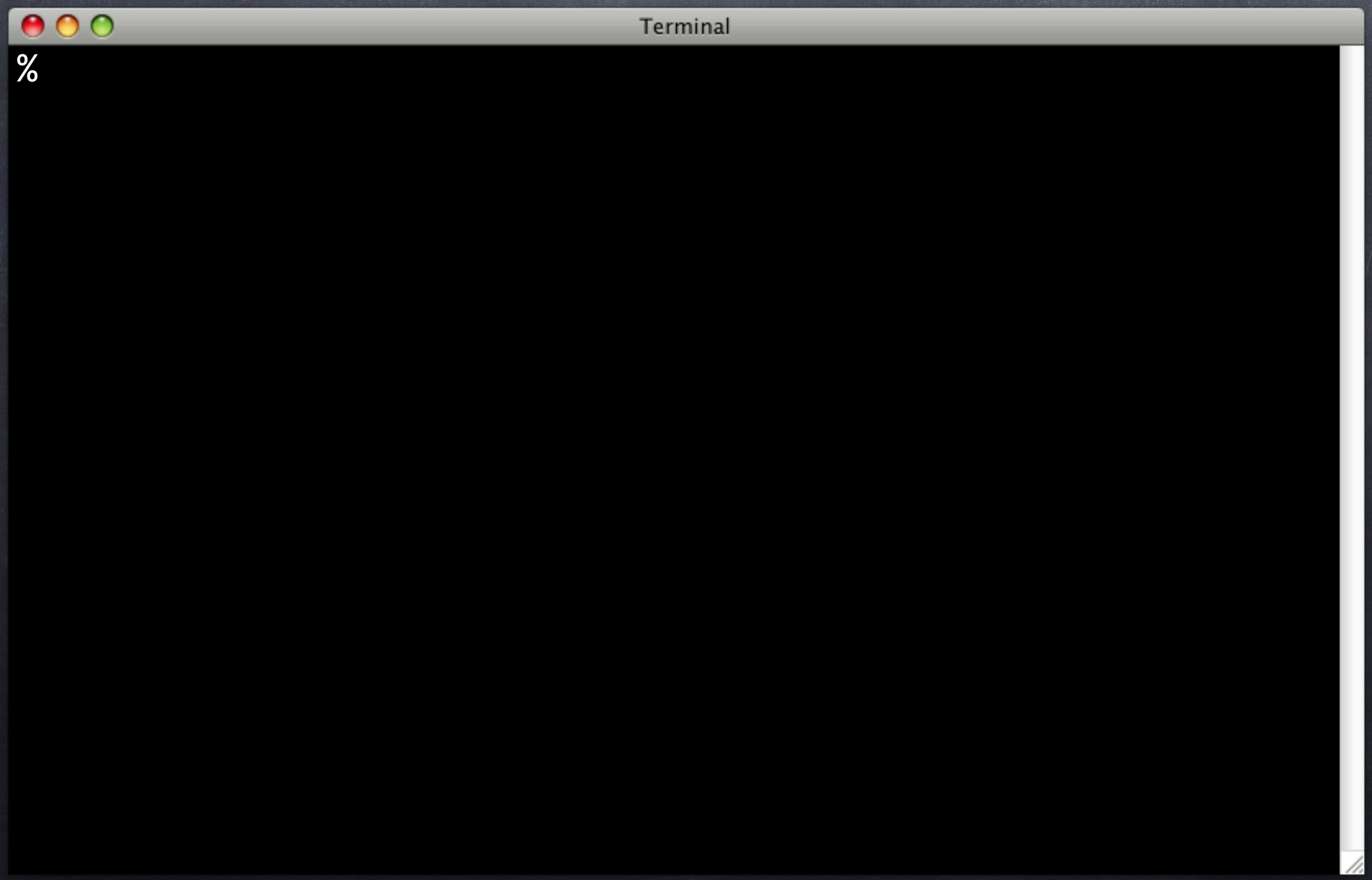
--:-- try.sql All (SQL[ansi])

Emacs

```sql
BEGIN;
SET search_path TO public, tap;
--SELECT plan(15);
SELECT * FROM no_plan();

SELECT can('{find_by_birthday}');
SELECT can_ok(
    'find_by_birthday',
    ARRAY['integer', 'integer', 'integer', 'integer', 'text']
);

-- Set up fixtures.
ALTER SEQUENCE users_user_id_seq RESTART 1;
INSERT INTO users (name, birthdate, birth_mon, birth_day, birth_year)
VALUES ('David', '1968-12-19', '12', '19', '1968'),
       ('Josh',  '1970-03-12', '03', '12', '1970'),
       ('Dan',   '1972-06-03', '6',  '3',  '1972'),
       ('Anna',  '2005-06-03', '06', '3',  '2005');

SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);

SELECT * FROM finish();
ROLLBACK;
```

--:-- try.sql All (SQL[ansi])

How We Doin'?



How We Doin'?

Terminal

```
% pg_prove -d try test_schema.sql
test_find_by_bday.sql .. ok
All tests successful.
Files=1, Tests=3, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

Emacs

```
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])----------

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

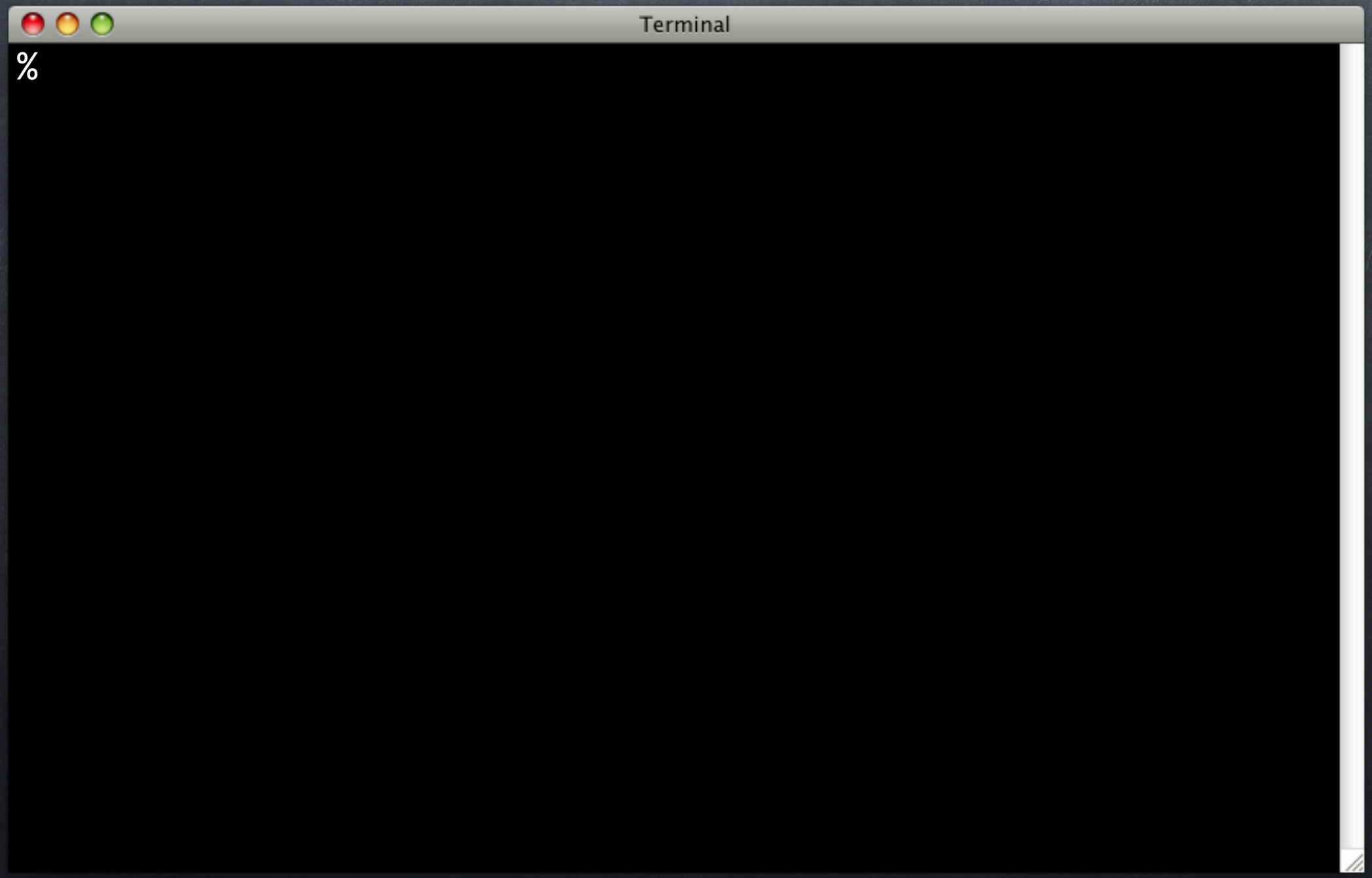
--:-- try.sql All (SQL[ansi])

Emacs

```sql
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 19, 12, NULL, NULL, 'active' ) ),
    ARRAY[1],
    'Should fetch one birthday for 12/19'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'active' ) ),
    ARRAY[3,4],
    'Should fetch two birthdays for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, 1, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one birthday for 3/6 OFFSET 1'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, 1, 'active' ) ),
    ARRAY[3],
    'Should fetch one birthday for 3/6 LIMIT 1'
);
UPDATE users SET state = 'inactive' WHERE user_id = 3;
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6 NULL, NULL, 'active' ) ),
    ARRAY[4],
    'Should fetch one active birthday for 3/6'
);
SELECT is(
    ARRAY( SELECT * FROM find_by_birthday( 3, 6, NULL, NULL, 'inactive' ) ),
    ARRAY[3],
    'Should fetch one inactive birthday for 3/6'
);
```

--:-- try.sql All (SQL[ansi])

Still Good...



Still Good...

Terminal

```
% pg_prove -d try test_schema.sql
test_find_by_bday.sql .. ok
All tests successful.
Files=1, Tests=8, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

NOW We Can Refactor

Let's Go with SQL

Let's Go with SQL

Emacs

```
CREATE OR REPLACE FUNCTION find_by_birthday(
    p_day    integer,
    p_mon    integer,
    p_offset integer,
    p_limit  integer,
    p_state  text
) RETURNS SETOF integer AS $$
    SELECT user_id
      FROM users
     WHERE state = COALESCE($5, 'active')
       AND EXTRACT(day   FROM birthdate) = $1
       AND EXTRACT(month FROM birthdate) = $2
     ORDER BY user_id
    OFFSET COALESCE( $3, NULL )
     LIMIT COALESCE( $4, NULL )
$$ LANGUAGE sql;
```

--:-- try.sql All (SQL[ansi])

Let's Go with SQL

Emacs

```
CREATE OR REPLACE FUNCTION find_by_birthday(
    p_day    integer,
    p_mon    integer,
    p_offset integer,
    p_limit  integer,
    p_state  text
) RETURNS SETOF integer AS $$
    SELECT user_id
      FROM users
     WHERE state = COALESCE($5, 'active')
       AND EXTRACT(day   FROM birthdate) = $1
       AND EXTRACT(month FROM birthdate) = $2
     ORDER BY user_id
    OFFSET COALESCE( $3, NULL )
     LIMIT COALESCE( $4, NULL )
$$ LANGUAGE sql;
```

--:-- try.sql All (SQL[ansi])

Let's Go with SQL

Emacs

```
CREATE OR REPLACE FUNCTION find_by_birthday(
    p_day    integer,
    p_mon    integer,
    p_offset integer,
    p_limit  integer,
    p_state  text
) RETURNS SETOF integer AS $$
    SELECT user_id
      FROM users
     WHERE state = COALESCE($5, 'active')
       AND EXTRACT(day   FROM birthdate) = $1
       AND EXTRACT(month FROM birthdate) = $2
     ORDER BY user_id
    OFFSET COALESCE( $3, NULL )
     LIMIT COALESCE( $4, NULL )
$$ LANGUAGE sql;
```

--:-- try.sql All (SQL[ansi])

Let's Go with SQL

Emacs

```
CREATE OR REPLACE FUNCTION find_by_birthday(
    p_day    integer,
    p_mon    integer,
    p_offset integer,
    p_limit  integer,
    p_state  text
) RETURNS SETOF integer AS $$
    SELECT user_id
      FROM users
     WHERE state = COALESCE($5, 'active')
       AND EXTRACT(day   FROM birthdate) = $1
       AND EXTRACT(month FROM birthdate) = $2
     ORDER BY user_id
    OFFSET COALESCE( $3, NULL )
     LIMIT COALESCE( $4, NULL )
$$ LANGUAGE sql;
```

--:-- try.sql All (SQL[ansi])

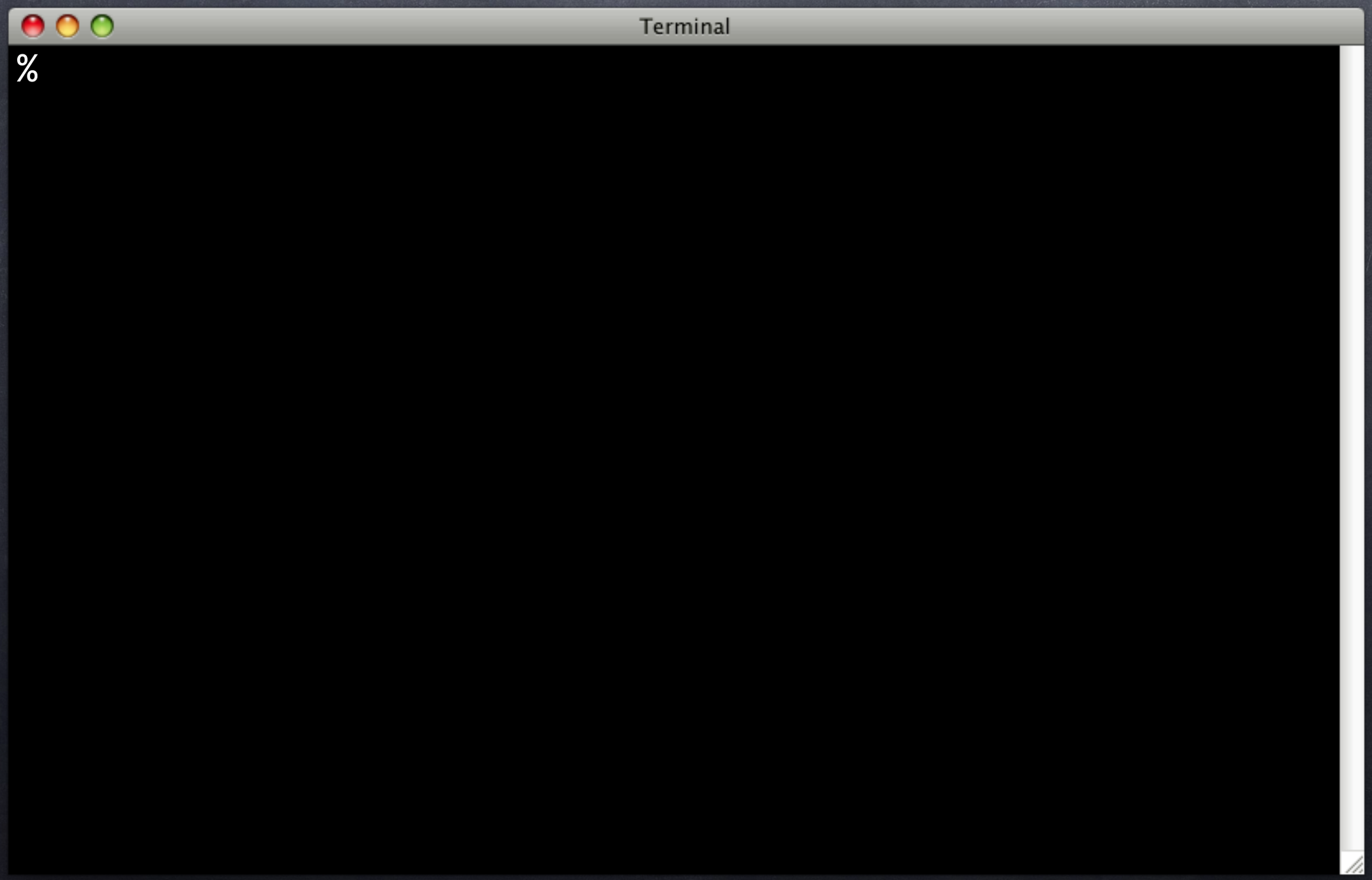
Let's Go with SQL

Emacs

```
CREATE OR REPLACE FUNCTION find_by_birthday(
    p_day    integer,
    p_mon    integer,
    p_offset integer,
    p_limit  integer,
    p_state  text
) RETURNS SETOF integer AS $$
    SELECT user_id
      FROM users
     WHERE state = COALESCE($5, 'active')
       AND EXTRACT(day   FROM birthdate) = $1
       AND EXTRACT(month FROM birthdate) = $2
     ORDER BY user_id
    OFFSET COALESCE( $3, NULL )
     LIMIT COALESCE( $4, NULL )
$$ LANGUAGE sql;
```

--:-- try.sql All (SQL[ansi])

And That's That



And That's That

Terminal

```
% pg_prove -d try test_schema.sql
test_find_by_bday.sql .. ok
All tests successful.
Files=1, Tests=8, 0 secs (0.02 usr + 0.01 sys = 0.03 CPU)
Result: PASS
%
```

Hell Yes!

Let's Review

Tests are for Finding Bugs

Tests are for Finding Bugs

- TDD not for finding bugs

Tests are for Finding Bugs

- TDD not for finding bugs
- TDD for sanity and consistency

Tests are for Finding Bugs

- TDD not for finding bugs
- TDD for sanity and consistency
- Tests prevent future bugs

Tests are Hard

Tests are Hard

- Good frameworks easy

Tests are Hard

- Good frameworks easy
- pgTAP provides lots of assertions

Tests are Hard

- Good frameworks easy
- pgTAP provides lots of assertions
- If you mean Hard to test interface:

Tests are Hard

- Good frameworks easy
- pgTAP provides lots of assertions
- If you mean Hard to test interface:
  - Red flag

Tests are Hard

- Good frameworks easy
- pgTAP provides lots of assertions
- If you mean Hard to test interface:
  - Red flag
  - Think about refactoring

Tests are Hard

- Good frameworks easy
- pgTAP provides lots of assertions
- If you mean Hard to test interface:
  - Red flag
  - Think about refactoring
  - If it's hard to test...

Tests are Hard

- Good frameworks easy
- pgTAP provides lots of assertions
- If you mean Hard to test interface:
  - Red flag
  - Think about refactoring
  - If it's hard to test...
  - It's hard to use

Never Find Relevant Bugs

Never Find Relevant Bugs

- Tests don't find bugs

Never Find Relevant Bugs

- Tests don't find bugs
- Test **PREVENT** bugs

Never Find Relevant Bugs

- Tests don't find bugs
- Test **PREVENT** bugs
- If your code doesn't work...

Never Find Relevant Bugs

- Tests don't find bugs
- Test **PREVENT** bugs
- If your code doesn't work...
- That failure is **RELEVANT**, no?

Time-Consuming

Time-Consuming

- Good frameworks easy to use

Time-Consuming

- Good frameworks easy to use
- Iterating between tests and code is natural

Time-Consuming

- Good frameworks easy to use
- Iterating between tests and code is natural
- Tests are as fast as your code

Time-Consuming

- Good frameworks easy to use
- Iterating between tests and code is natural
- Tests are as fast as your code
- Not as time-consuming as bug hunting

Time-Consuming

- Good frameworks easy to use
- Iterating between tests and code is natural
- Tests are as fast as your code
- Not as time-consuming as bug hunting
- When no tests, bugs repeat themselves

Time-Consuming

- Good frameworks easy to use
- Iterating between tests and code is natural
- Tests are as fast as your code
- Not as time-consuming as bug hunting
- When no tests, bugs repeat themselves
- And are harder to track down

Time-Consuming

- Good frameworks easy to use
- Iterating between tests and code is natural
- Tests are as fast as your code
- Not as time-consuming as bug hunting
- When no tests, bugs repeat themselves
- And are harder to track down
- Talk about a time sink!

Running Tests is Slow

Running Tests is Slow

- Test what you're working on

Running Tests is Slow

- Test what you're working on
- Set up automated testing for everything else

Running Tests is Slow

- Test what you're working on
- Set up automated testing for everything else
- Pay attention to automated test failures

For Inexperienced Developers

For Inexperienced Developers

- I've been programming for 10 years

For Inexperienced Developers

- I've been programming for 10 years
- I have no idea what I was thinking a year ago

For Inexperienced Developers

- I've been programming for 10 years
- I have no idea what I was thinking a year ago
- Tests make maintenance a breeze

For Inexperienced Developers

- I've been programming for 10 years
- I have no idea what I was thinking a year ago
- Tests make maintenance a breeze
- They give me the confidence to make changes without fearing the consequences

For Inexperienced Developers

- I've been programming for 10 years
- I have no idea what I was thinking a year ago
- Tests make maintenance a breeze
- They give me the confidence to make changes without fearing the consequences
- Tests represent **FREEDOM** from the tyranny of fragility and inconsistency

Unnecessary for Simple Code

Unnecessary for Simple Code

- I copied fib() from a Perl library

Unnecessary for Simple Code

- I copied fib() from a Perl library
- It was dead simple

Unnecessary for Simple Code

- I copied fib() from a Perl library
- It was dead simple
- And it was still wrong

Unnecessary for Simple Code

- I copied fib() from a Perl library
- It was dead simple
- And it was still wrong
- Tests keep even the simplest code working

Best for Fragile Code

Best for Fragile Code

- All code is fragile

Best for Fragile Code

- All code is fragile
- Tests make code ROBUST

Best for Fragile Code

- All code is fragile
- Tests make code ROBUST
- Add regression tests for bugs found by

Best for Fragile Code

- All code is fragile
- Tests make code ROBUST
- Add regression tests for bugs found by
  - Integration tests

Best for Fragile Code

- All code is fragile
- Tests make code ROBUST
- Add regression tests for bugs found by
  - Integration tests
  - QA department

Best for Fragile Code

- All code is fragile
- Tests make code ROBUST
- Add regression tests for bugs found by
  - Integration tests
  - QA department
  - Your users

Users Test our Code

Users Test our Code

- Talk about fragility

Users Test our Code

- Talk about fragility
- Staging servers never work

Users Test our Code

- Talk about fragility
- Staging servers never work
- QA departments are disappearing

Users Test our Code

- Talk about fragility
- Staging servers never work
- QA departments are disappearing
- Users don't want to see bugs

Users Test our Code

- Talk about fragility
- Staging servers never work
- QA departments are disappearing
- Users don't want to see bugs
- Find ways to test your code

Users Test our Code

- Talk about fragility
- Staging servers never work
- QA departments are disappearing
- Users don't want to see bugs
- Find ways to test your code
- Users avoid fragile applications

It's a Private Function

It's a Private Function

- It still needs to work

It's a Private Function

- It still needs to work
- It still needs to always work

It's a Private Function

- It still needs to work
- It still needs to always work
- Don't reject glass box testing

It's a Private Function

- It still needs to work
- It still needs to always work
- Don't reject glass box testing
- Make sure that ALL interfaces work

Application Tests are Sufficient

Application Tests are Sufficient

- App tests should connect as as app user

Application Tests are Sufficient

- App tests should connect as as app user
- May well be security limitations for the app

Application Tests are Sufficient

- App tests should connect as as app user
- May well be security limitations for the app
  - Access only to functions

Application Tests are Sufficient

- App tests should connect as as app user
- May well be security limitations for the app
  - Access only to functions
- Apps cannot adequately test the database

Application Tests are Sufficient

- App tests should connect as as app user
- May well be security limitations for the app
  - Access only to functions
- Apps cannot adequately test the database
- Database tests should test the database

Application Tests are Sufficient

- App tests should connect as as app user
- May well be security limitations for the app
  - Access only to functions
- Apps cannot adequately test the database
- Database tests should test the database
- Application tests should test the application

Tests Prove Nothing

Tests Prove Nothing

- This is not a math equation

Tests Prove Nothing

- This is not a math equation
- This is about:

Tests Prove Nothing

- This is not a math equation
- This is about:
  - consistency

Tests Prove Nothing

- This is not a math equation
- This is about:
  - consistency
  - stability

Tests Prove Nothing

- This is not a math equation
- This is about:
  - consistency
  - stability
  - robusticity

Tests Prove Nothing

- This is not a math equation
- This is about:
  - consistency
  - stability
  - robusticity
- If a test fails, it has proved a failure

Tests Prove Nothing

- This is not a math equation
- This is about:
  - consistency
  - stability
  - robusticity
- If a test fails, it has proved a failure
- Think Karl Popper

Tests are for Stable Code

Tests are for Stable Code

- How does it become stable?

Tests are for Stable Code

- How does it become stable?
- Tests the fastest route

Tests are for Stable Code

- How does it become stable?
- Tests the fastest route
- Ensure greater stability over time

Tests are for Stable Code

- How does it become stable?
- Tests the fastest route
- Ensure greater stability over time
- TDD help with working through issues

Tests are for Stable Code

- How does it become stable?
- Tests the fastest route
- Ensure greater stability over time
- TDD help with working through issues
- TDD helps thinking through interfaces

Tests are for Stable Code

- How does it become stable?
- Tests the fastest route
- Ensure greater stability over time
- TDD help with working through issues
- TDD helps thinking through interfaces
- Tests encourage experimentation

I Really Like Detroit

I Really Like Detroit

- I can't help you

What're You Waiting For?

What're You Waiting For?

- pgTAP: http://pgtap.projects.postgresql.org

What're You Waiting For?

- pgTAP: http://pgtap.projects.postgresql.org
- pgUnit: http://en.dklab.ru/lib/dklab_pgunit

What're You Waiting For?

- pgTAP: http://pgtap.projects.postgresql.org
- pgUnit: http://en.dklab.ru/lib/dklab_pgunit
- EpicTest: http://www.epictest.org

What're You Waiting For?

- pgTAP: http://pgtap.projects.postgresql.org
- pgUnit: http://en.dklab.ru/lib/dklab_pgunit
- EpicTest: http://www.epictest.org
- pg_regress

Start writing tests

Increase consistency

Improve stability

Save time

Free yourself

and...

Kick ass.

Thank You

Unit Test Your Database!

David E. Wheeler
PostgreSQL Experts, Inc.

PGCon, May 21, 2009

PGX
POSTGRESQL
EXPERTS, INC.