Bug Hunting For Dummies

Antonio Cuni

EuroPython 2013

July 2, 2013

About me

- PyPy core dev
- `pdb++`, `fancycompleter`, ...
- Consultant, trainer
- `http://antocuni.eu`

About this talk

If debugging is the process of removing bugs, then programming must be the process of putting them in.
(Dykstra's Observation)

- 80% of development is spent in debugging
- 80% of debugging is spent in **finding** the bugs
  - Bug hunting!

- No silver bullet
- The mindset of the bug hunter
- Examples of techniques I use

About this talk

If debugging is the process of removing bugs, then programming must be the process of putting them in.
(Dykstra's Observation)

- 80% of development is spent in debugging
- 80% of debugging is spent in **finding** the bugs
  - Bug hunting!

- No silver bullet
- The mindset of the bug hunter
- Examples of techniques I use

About this talk

If debugging is the process of removing bugs, then programming must be the process of putting them in.
(Dykstra's Observation)

- 80% of development is spent in debugging
- 80% of debugging is spent in **finding** the bugs
  - Bug hunting!

- No silver bullet
- The mindset of the bug hunter
- Examples of techniques I use

What is a bug?

- (Un)expected behaviour of a program
  - Crash
  - Incorrect result
  - Memory leak
  - Performance problem
- Categories
  - Deterministic
  - Undeterministic
  - Heisenbugs

What is a bug?

- (Un)expected behaviour of a program
  - Crash
  - Incorrect result
  - Memory leak
  - Performance problem
- Categories
  - Deterministic
  - Undeterministic
  - Heisenbugs

Scenario

- Big project
- Lot of code, lots of people, several years of development
- Complex relations in the source code (e.g. PyPy :))
- VPBR (Very Precise Bug Report)
  - *“the program does not work!”*
- “Big”, “lot” and “complex” are subjective
- There will always be a level of complexity which you can’t understand immediately.

Scenario

- Big project
- Lot of code, lots of people, several years of development
- Complex relations in the source code (e.g. PyPy :))
- VPBR (Very Precise Bug Report)
  - *“the program does not work!”*
- “Big”, “lot” and “complex” are subjective
- There will always be a level of complexity which you can’t understand immediately.

Scenario

- Big project
- Lot of code, lots of people, several years of development
- Complex relations in the source code (e.g. PyPy :))
- VPBR (Very Precise Bug Report)
  - *“the program does not work!”*
- “Big”, “lot” and “complex” are subjective
- There will always be a level of complexity which you can’t understand immediately.

Naïve approach

- Guess where is the problem
- Locate the related source code
- Repeat:
  - try to understand the mess of the source code
  - (optional: inspect in a debugger)
  - fix&try

- Very fast if the bug is simple
- Little chance of success if the bug is complex
  - The world stop to make any sense

Naïve approach

- Guess where is the problem
- Locate the related source code
- Repeat:
  - try to understand the mess of the source code
  - (optional: inspect in a debugger)
  - fix&try
- Very fast if the bug is simple
- Little chance of success if the bug is complex
  - The world stop to make any sense

Mindset

- There MUST be an explanation
- No divinity or god is against you
- Your assumptions might be wrong
- The compiler/library/O.S. is probably correct
  - Unless it's not :)

Mindset

- There MUST be an explanation
- No divinity or god is against you
- Your assumptions might be wrong
- The compiler/library/O.S. is probably correct
  - Unless it's not :)

General approach

(not necessarily in this order)

0. (from the Zen of Python): Refuse the temptation to guess
1. Reproduce the bug
2. Automatize the run (you are going to run it **many** times)
3. Find the smallest test case which fails
4. Spot the problem.
5. Understand the problem
6. **Make predictions**, run, repeat. Fixed.

- Goal: Understand, **then** fix

1. Reproduce the bug

- Might be tricky sometimes
- Try to reproduce it locally
  - “but it works on my machine!”
- Write a test
- Pay attention to all the possible variables
  - Operating System
  - Version of program&libraries
  - CPU, 32/64 bit
  - Workload, RAM size
  - Network latency/bandwith
  - Localization
  - Phase of the moon
  - ..., plus any combination of the above

1. Reproduce the bug

- Might be tricky sometimes
- Try to reproduce it locally
  - “but it works on my machine!”
- Write a test
- Pay attention to all the possible variables
  - Operating System
  - Version of program&libraries
  - CPU, 32/64 bit
  - Workload, RAM size
  - Network latency/bandwith
  - Localization
  - Phase of the moon
  - ..., plus any combination of the above

2. Automation

- “One click away bugs”
- Avoid manual input from the user
- Example: GUI with a “load” button
  - --> small program to call the event handler directly
- Example: web application
  - --> small program which sends the “right” HTTP requests
- At worst: mouse automation (autopy, pywinauto)
- Write a test

3. Reduction

- Goal: smallest possible program which still fail
- Example: crash in an HTML parser
- Reduce the data
  - Remove some of the tags of the offending HTML document
  - Check whether it still fails
  - Repeat
- Reduce the code
  - Remove the code for handling malformed HTML
  - If it still fails, the problem is somewhere else
  - If it stops failing, the problem is there
- Write a test

4. Spot the problem

- Bigger reduction --> easier hunting
- If it's still too complex
  - step by step in a debugger
  - print/logging/tracing
  - strace, ltrace

5-6. Understand & fix

- Refuse the temptation to guess
- Fix only **after** you understood the problem
- Make predictions
  - (a.k.a: “the scientific method”, G. Galilei, 1638 ca)
  - Make a change
  - Have precise expectations on the output
  - Verify
- The mind adapt its view of the world to what you observe.
- **Write a test** (if you didn’t yet)
  - Almost for free once you have reduced&automated
  - Fail before, pass after the fix

Excursus: benefits of TDD (1)

- Write a (failing) test before implementing a feature
- Write a (failing) test before fixing a bug
- Check you didn't introduce any other bug: rerun the entire testsuite
- Commit!
- Regression: "something stopped working"
- A test fails, but used to pass

Excursus: benefits of TDD(2)

1. `hg bisect --reset; hg bisect --bad`
2. `hg up -r some-rev-where-the-test-passed`
3. check whether the test passes
   - *`hg bisect --good`* ***OR***
   - *`hg bisect --bad`*
4. goto 3

- OR: `hg bisect -c py.test test_myfile.py -k my_failing_test`
- `git bisect` works similarly

Excursus: benefits of TDD(2)

1. `hg bisect --reset; hg bisect --bad`
2. `hg up -r some-rev-where-the-test-passed`
3. check whether the test passes
   - *`hg bisect --good`* ***OR***
   - *`hg bisect --bad`*
4. goto 3

- OR: `hg bisect -c py.test test_myfile.py -k my_failing_test`
- `git bisect` works similarly

Real world example

- PyPy `_fastjson` decoder
- “Large” document crashes in the middle
- “Simple&fast” approach
  - Locate the error message
  - Look around, put a pdb, try to guess
  - No way
- “General approach”
  - Reduce the data!
  - Write a test
  - (fix)

Real world example

- PyPy `_fastjson` decoder
- “Large” document crashes in the middle
- “Simple&fast” approach
  - Locate the error message
  - Look around, put a pdb, try to guess
  - No way
- “General approach”
  - Reduce the data!
  - Write a test
  - (fix)

Don’t assume the assumptions

When you have eliminated the impossible, whatever remains, however improbable, must be the truth (Sherlock Holmes, Sir Arthur Conan Doyle, “The Sign of Four”)

- Be prepared to question everything
- Challenge your assumptions
  - put a `print` to make sure a function is called
  - put `assert` everywhere
  - write passing tests to check that the world behave as you expect
  - make sure you are editing the right file!

The “XXX” technique

- Your fix some code, but the system ignores you
- Put an `XXX` in the code and check that it breaks
- .pyc files
- the module is imported from somewhere else:
  - a different checkout
  - site-packages/ vs your development dir
- two functions with the same name
- `print __file__`
- `print mymodule.__file__`
- `print myfunction.__module__`

Useful tools

- Debuggers:
  - `pdb`, `pdb++`, `pudb`, `ipdb`
  - `pip install pdbpp` (unix-only, sorry)
  - IDE debuggers (PyCharm, Wing IDE, etc.)
- Breakpoints and step-by-step execution
  - `import pdb;pdb.set_trace()`

Post-mortem debugging

- `py.test --pdb test_myprogram.py`
- `python -m pdb myprogram.py`

Automatic post-mortem pdb

```
import sys
import traceback
import pdb

def start_pdb(type, value, tb):
    traceback.print_exception(type, value, tb)
    print
    pdb.pm()

sys.excepthook = start_pdb
```

Post-mortem debugging

- `py.test --pdb test_myprogram.py`
- `python -m pdb myprogram.py`

Automatic post-mortem pdb

```
import sys
import traceback
import pdb

def start_pdb(type, value, tb):
    traceback.print_exception(type, value, tb)
    print
    pdb.pm()

sys.excepthook = start_pdb
```

Eaten exceptions (1)

- Exceptions caugth before they reach you
- No chance to enter post-mortem
  - because there is no mortem :)
- E.g.: logging

somewhere deep in the code...

```
try:
    do_something()
except Exception, e:
    logfile.write('An error occoured: %s' % e)
```

Eaten exceptions (2)

somewhere deep in the code...

```
try:
    do_something()
except Exception, e:
    # this will open a pdb at the point where
    # the exceptions was originally raised
    pdb.post_mortem(sys.exc_info()[2])
    logfile.write('An error occoured: %s' % e)
```

with pdb++

```
try:
    do_something()
except Exception, e:
    # equivalent to the above
    pdb.xpm()
    logfile.write('An error occoured: %s' % e)
```

Eaten exceptions (2)

somewhere deep in the code...

```
try:
    do_something()
except Exception, e:
    # this will open a pdb at the point where
    # the exceptions was originally raised
    pdb.post_mortem(sys.exc_info()[2])
    logfile.write('An error occoured: %s' % e)
```

with pdb++

```
try:
    do_something()
except Exception, e:
    # equivalent to the above
    pdb.xpm()
    logfile.write('An error occoured: %s' % e)
```

Eaten exceptions (3)

- Don't eat exceptions unless you really have to

NEVER do it

```
try:
    do_something()
except:
    pass
```

Logging/tracing vs manual step-by-step

- sometimes step-by-step is not always applicable
  - `http://antocuni.eu/misc/tracker.txt`

Trace calls

```
call = CallTracker()

@call.track
def foo(x, y):
    return bar(x) + baz(y)

@call.track
def bar(a):
    call.log('a ==', a)
    return a*2

@call.track
def baz(b):
    return b*3

print foo(10, 20)
```

Trace calls

```
class CallTracker(object):
    def __init__(self):
        self.level = 0

    def log(self, *args):
        print('    ' * self.level, *args)

    def exit(self, f):
        self.level -= 1

    def enter(self, f):
        self.log('entering', f)
        self.level += 1

    def track(self, fn):
        def newfn(*args, **kwds):
            self.enter(fn.__name__)
            try:
                return fn(*args, **kwds)
            finally:
                self.exit(fn.__name__)
        return newfn
```

Logging/tracing vs manual step-by-step

- sometimes step-by-step is not always applicable
  - `http://antocuni.eu/misc/tracker.txt`

Trace calls

```
call = CallTracker()

@call.track
def foo(x, y):
    return bar(x) + baz(y)

@call.track
def bar(a):
    call.log('a ==', a)
    return a*2

@call.track
def baz(b):
    return b*3

print foo(10, 20)
```

Trace calls

```
class CallTracker(object):
    def __init__(self):
        self.level = 0

    def log(self, *args):
        print('   ' * self.level, *args)

    def exit(self, f):
        self.level -= 1

    def enter(self, f):
        self.log('entering', f)
        self.level += 1

    def track(self, fn):
        def newfn(*args, **kwds):
            self.enter(fn.__name__)
            try:
                return fn(*args, **kwds)
            finally:
                self.exit(fn.__name__)
        return newfn
```

Conditional breakpoints

```
if something:
    import pdb;pdb.set_trace()
```

- During the debugging you are allowed to cheat:
  - E.g.: check whether a string containing “foobar” is in a list
  - `if "foobar" in repr(myobj)`
- Combine it with logging
- E.g.: stop only just before the crash
- _fastjson demo

Exploit Python dynamicity (1)

pdb on setattr

```
class Foo(object):

    def __setattr__(self, name, value):
        if name == 'x' and value == 42:
            import pdb;pdb.set_trace()
        object.__setattr__(self, name, value)

f = Foo()
f.y = 123
f.x = 456
f.x = 42 # PDB!
```

Exploit Python dynamicity (2)

using pdb++

```
import pdb # this is actually pdb++

def is_42(obj, value):
    return value == 42

@pdb.break_on_setattr('x', condition=is_42)
class Foo(object):
    pass

f = Foo()
f.y = 123
f.x = 456
f.x = 42 # PDB!
```

Exploit Python dynamicity (3)

- Real world example
- Where the hell is the Django code to locate a template?!?

break on file open

```
import __builtin__

original_open = open
def myopen(filename, *args):
    if 'passwd' in filename:
        import pdb;pdb.set_trace()
    return original_open(filename, *args)

__builtin__.open = myopen
__builtin__.file = myopen

print open(__file__).read()
print open('/etc/passwd').read()
```

Exploit Python dynamicity (4)

- Useful when you don't know/find who prints a certain message

break on stdout

```
import sys

class MyStdout(object):
    def __init__(self, out):
        self.out = out

    def write(self, s):
        if 'i == 100' in s:
            import pdb;pdb.set_trace()
        self.out.write(s)

sys.stdout = MyStdout(sys.stdout)

for i in range(200):
    print 'i == %d' % i
```

Non-deterministic bugs (1)

- The bug does not show always
- Might depend on the phase of the moon
- In C: unitialized memory, freed pointers, etc.
- Much less common in Python

- Dictionary order
- C extensions
- ctypes, cffi & co.
- Threading, race conditions

Non-deterministic bugs (1)

- The bug does not show always
- Might depend on the phase of the moon
- In C: unitialized memory, freed pointers, etc.
- Much less common in Python

- Dictionary order
- C extensions
- ctypes, cffi & co.
- Threading, race conditions

Non-deterministic bugs (2)

- You cannot be sure to have solved the problem
- Key: make it deterministic
  - Run it N times (remember automation?)
  - Use a larger input
  - Try to change the conditions (e.g. free memory, CPU load)
  - Put random sleeps/busy loops in the threads
  - Allocate N big objects at the start
- Exploit low level tools
  - gdb watch
  - valgrind

Heisenbugs

- The program crashes
- As soon as you modify for inspection, it works
- Yes, some divinity might be against you

- Leaving it modified is NOT the solution! :)
- Inspect without modify
  - use print/logging instead of pdb.set_trace()
  - inspect from gdb
  - strace, ltrace
- No general solution, sorry :-(

Heisenbugs

- The program crashes
- As soon as you modify for inspection, it works
- Yes, some divinity might be against you

- Leaving it modified is NOT the solution! :)
- Inspect without modify
  - use print/logging instead of pdb.set_trace()
  - inspect from gdb
  - strace, ltrace
- No general solution, sorry :-(

Contacts, Q&A

- twitter: @antocuni
- Available for consultancy & training:
  - `http://antocuni.eu`
  - info@antocuni.eu
- Any question?