Introduction to PyPy

Antonio Cuni

BigDive 2012

October 22 2012

What is PyPy?

- PyPy
  - started in 2003
  - Open Source, partially funded by EU and others
  - framework for fast dynamic languages
  - **Python implementation**
- as a Python dev, you care about the latter

Python in Python

- Actually: Python in **RPython**
- Restricted Python
  - Statically typed subset
  - never designed to be user friendly
  - still better than C/Java/C# in lots of aspects
  - “we write RPython so you don’t have to” (cit.)
- RPython : PyPy = C : CPython ...
- ... Java : Jython = C# : IronPython

RPython

- Run RPython programs on top of CPython
  - isn't it damn slow? Yes.
- Compile RPython programs to C
  - this is where the magic happens

RPython

- Run RPython programs on top of CPython
  - isn’t it damn slow? Yes.
- Compile RPython programs to C
  - this is where the magic happens

PyPy: Software archeology

- Around since 2003
- (advertised as) production ready since December 2010
  - release 1.4
- Funding
  - EU FP6 programme
  - Eurostars programme
  - donations
  - ...

PyPy 1.9: current status

- Faster
  - **1.7x** than 1.5 (a year ago)
  - **2.2x** than 1.4
  - **5.5x** than CPython
- Implements Python 2.7.2
- Many more “PyPy-friendly” programs
- Packaging
  - Debian, Ubuntu, Fedora, Homebrew, Gentoo, ArchLinux, ...
  - Windows (32bit only), OS X
- C extension compatibility
  - runs (big part of) **PyOpenSSL** and **lxml**

PyPy features

- JIT
  - automatically generated
  - complete/correct by construction
  - multiple backends: x86-32, x86-64, ARM
- Stackless
  - not yet integrated with the JIT (in-progress)
- cpyext
  - CPython C-API compatibility layer
  - not always working
  - often working: wxPython, PIL, cx_Oracle, mysqldb, pycairo, ...
- compact instances (as using `__slots__`)

PyPy features

- JIT
  - automatically generated
  - complete/correct by construction
  - multiple backends: x86-32, x86-64, ARM
- Stackless
  - not yet integrated with the JIT (in-progress)
- cpyext
  - CPython C-API compatibility layer
  - not always working
  - often working: wxPython, PIL, cx_Oracle, mysqldb, pycairo, ...
- compact instances (as using `__slots__`)

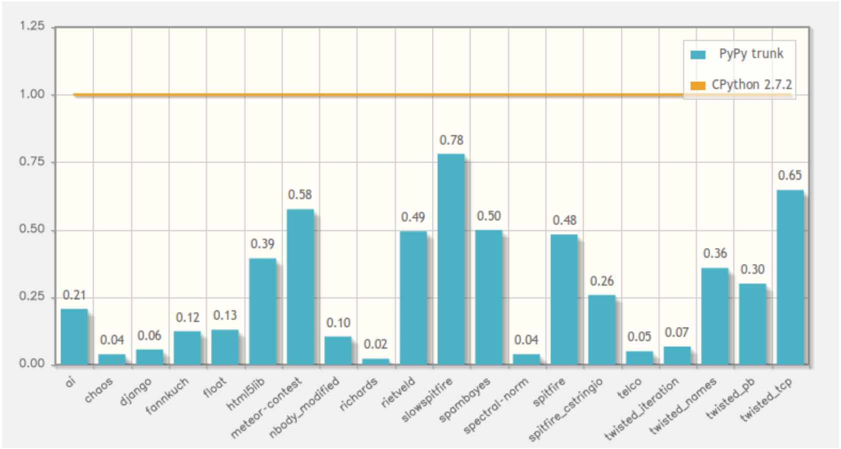
PyPy features

- JIT
  - automatically generated
  - complete/correct by construction
  - multiple backends: x86-32, x86-64, ARM
- Stackless
  - not yet integrated with the JIT (in-progress)
- cpyext
  - CPython C-API compatibility layer
  - not always working
  - often working: wxPython, PIL, cx_Oracle, mysqldb, pycairo, ...
- compact instances (as using `__slots__`)

PyPy features

- JIT
  - automatically generated
  - complete/correct by construction
  - multiple backends: x86-32, x86-64, ARM
- Stackless
  - not yet integrated with the JIT (in-progress)
- cpyext
  - CPython C-API compatibility layer
  - not always working
  - often working: wxPython, PIL, cx_Oracle, mysqldb, pycairo, ...
- compact instances (as using `__slots__`)

Speed


1.25
1.00
0.75
0.50
0.25
0.00
PyPy trunk
CPython 2.7.2
0.21
0.04
0.06
0.12
0.13
0.39
0.58
0.10
0.02
0.49
0.78
0.50
0.04
0.48
0.26
0.05
0.07
0.36
0.30
0.65
ai
chaos
django
fannkuch
float
html5lib
meteor-contest
nbody_modified
richards
rietveld
slowspitfire
spambayes
spectral-norm
spitfire
spitfire_cstringio
telco
twisted_iteration
twisted_names
twisted_pb
twisted_tcp

Differences with CPython

- GC: not reference counting
  - `__del__`, `weakref`, etc.

refcounting

```python
def foo():
    f = open('/tmp/foo.txt')
    f.write('hello')
```

correct way

```python
def foo():
    with open('/tmp/foo.txt') as f:
        f.write('hello')
```

Differences with CPython

- GC: not reference counting
  - `__del__`, `weakref`, etc.

refcounting

```
def foo():
    f = open('/tmp/foo.txt')
    f.write('hello')
```

correct way

```
def foo():
    with open('/tmp/foo.txt') as f:
        f.write('hello')
```

Real world use case (1)

- LWN's gitdm
  - `http://lwn.net/Articles/442268/`
  - data mining tool
  - reads the output of `git log`
  - generate kernel development statistics
- Performance
  - CPython: 63 seconds
  - PyPy: **21 seconds**

`lwn.net`

[...] PyPy is ready for prime time; it implements the (Python 2.x) language faithfully, and it is fast.

Real world use case (1)

- LWN's gitdm
  - `http://lwn.net/Articles/442268/`
  - data mining tool
  - reads the output of `git log`
  - generate kernel development statistics
- Performance
  - CPython: 63 seconds
  - PyPy: **21 seconds**

`lwn.net`

[...] PyPy is ready for prime time; it implements the (Python 2.x) language faithfully, and it is fast.

Real world use case (1)

- LWN's gitdm
  - `http://lwn.net/Articles/442268/`
  - data mining tool
  - reads the output of `git log`
  - generate kernel development statistics
- Performance
  - CPython: 63 seconds
  - PyPy: **21 seconds**

`lwn.net`

[...] PyPy is ready for prime time; it implements the (Python 2.x) language faithfully, and it is fast.

Real world use case (2)

- **MyHDL**: VHDL-like language written in Python
  - `http://www.myhdl.org/doku.php/performance`
  - (now) competitive with “real world” VHDL and Verilog simulators

myhdl.org

[...] the results are spectacular. By simply using a different interpreter, our simulations run 6 to 12 times faster.

Real world use case (2)

- **MyHDL**: VHDL-like language written in Python
  - `http://www.myhdl.org/doku.php/performance`
  - (now) competitive with “real world” VHDL and Verilog simulators

`myhdl.org`

[...] the results are spectacular. By simply using a different interpreter, our simulations run 6 to 12 times faster.

Real world use case (3)

- Translating PyPy itself
- Huge, complex piece of software
- All possible (and impossible :-)) kinds of dynamic and metaprogrammig tricks
- ~2.5x faster with PyPy
- (slow warm-up phase, though)
- Ouroboros!

Real world use case (4)

- Your own application
- Try PyPy, it might be worth it

Not convinced yet?

Real time edge detection

```
def sobeldx(img):
  res = img.clone(typecode='d')
  for p in img.pixeliter():
      res[p] = (-1.0 * img[p + (-1,-1)] +
                 1.0 * img[p + ( 1,-1)] +
                -2.0 * img[p + (-1, 0)] +
                 2.0 * img[p + ( 1, 0)] +
                -1.0 * img[p + (-1, 1)] +
                 1.0 * img[p + ( 1, 1)]) / 4.0
  return res
...
...
```

Live demo

Is Python slow?

- ~~Python is slow~~
- Python is hard to optimize

- Huge stack of layers over the bare metal
- Abstraction has a cost (... or not?)

Is Python slow?

- ~~Python is slow~~
- Python is hard to optimize

- Huge stack of layers over the bare metal
- Abstraction has a cost (... or not?)

Is Python slow?

- ~~Python is slow~~
- Python is hard to optimize

- Huge stack of layers over the bare metal
- Abstraction has a cost (... or not?)

Python is complicated

How `a + b` works (simplified!):

- look up the method `__add__` on the type of a
- if there is one, call it
- if it returns NotImplemented, or if there is none, look up the method `__radd__` on the type of b
- if there is one, call it
- if there is none, or we get `NotImplemented` again, raise an exception `TypeError`

Python is a mess

How `obj.attr` or `obj.method()` works:

- ...
- no way to write it down in just one slide

Python is a mess

How `obj.attr` or `obj.method()` works:

- ...
- no way to write it down in just one slide

Killing the abstraction overhead

Python

```python
class Point(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __add__(self, q):
    if not isinstance(q, Point):
      raise TypeError
    x1 = self.x + q.x
    y1 = self.y + q.y
    return Point(x1, y1)

def main():
  p = Point(0.0, 0.0)
  while p.x < 2000.0:
    p = p + Point(1.0, 0.5)
  print p.x, p.y
```

C

```c
#include <stdio.h>









 int main() {
     float px = 0.0, py = 0.0;
     while (px < 2000.0) {
         px += 1.0;
         py += 0.5;
     }
     printf("%f %f\n", px, py);
 }
```

Killing the abstraction overhead

Python

```
class Point(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __add__(self, q):
    if not isinstance(q, Point):
      raise TypeError
    x1 = self.x + q.x
    y1 = self.y + q.y
    return Point(x1, y1)

def main():
  p = Point(0.0, 0.0)
  while p.x < 2000.0:
    p = p + Point(1.0, 0.5)
  print p.x, p.y
```

C

```
#include <stdio.h>









 int main() {
     float px = 0.0, py = 0.0;
     while (px < 2000.0) {
         px += 1.0;
         py += 0.5;
     }
     printf("%f %f\n", px, py);
 }
```

Pointless optimization techniques

```python
#
for item in some_large_list:
    self.meth(item)
```

```python
meth = self.meth
for item in some_large_list:
    meth(item)
```

```python
def foo():
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```python
def foo(abs=abs):
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```python
#

[i**2 for i in range(100)]
```

```python
from itertools import *
list(imap(pow, count(0),
          repeat(2, 100)))
```

```python
for i in range(large_number):
    ...
```

```python
for i in xrange(large_number):
    ...
```

```python
class A(object):
    pass
```

```python
class A(object):
    __slots__ = ['a', 'b', 'c']
```

Pointless optimization techniques

```
#
for item in some_large_list:
    self.meth(item)
```

```
meth = self.meth
for item in some_large_list:
    meth(item)
```

```
def foo():
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```
def foo(abs=abs):
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```
#

[i**2 for i in range(100)]
```

```
from itertools import *
list(imap(pow, count(0),
          repeat(2, 100)))
```

```
for i in range(large_number):
    ...
```

```
for i in xrange(large_number):
    ...
```

```
class A(object):
    pass
```

```
class A(object):
    __slots__ = ['a', 'b', 'c']
```

Pointless optimization techniques

```
#
for item in some_large_list:
    self.meth(item)
```

```
meth = self.meth
for item in some_large_list:
    meth(item)
```

```
def foo():
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```
def foo(abs=abs):
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```
#

[i**2 for i in range(100)]
```

```
from itertools import *
list(imap(pow, count(0),
          repeat(2, 100)))
```

```
for i in range(large_number):
    ...
```

```
for i in xrange(large_number):
    ...
```

```
class A(object):
    pass
```

```
class A(object):
    __slots__ = ['a', 'b', 'c']
```

Pointless optimization techniques

```
#
for item in some_large_list:
    self.meth(item)
```

```
meth = self.meth
for item in some_large_list:
    meth(item)
```

```
def foo():
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```
def foo(abs=abs):
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```
#

[i**2 for i in range(100)]
```

```
from itertools import *
list(imap(pow, count(0),
              repeat(2, 100)))
```

```
for i in range(large_number):
    ...
```

```
for i in xrange(large_number):
    ...
```

```
class A(object):
    pass
```

```
class A(object):
    __slots__ = ['a', 'b', 'c']
```

Pointless optimization techniques

```python
#
for item in some_large_list:
    self.meth(item)
```

```python
meth = self.meth
for item in some_large_list:
    meth(item)
```

```python
def foo():
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```python
def foo(abs=abs):
    res = 0
    for item in some_large_list:
        res = res + abs(item)
    return res
```

```python
#

[i**2 for i in range(100)]
```

```python
from itertools import *
list(imap(pow, count(0),
          repeat(2, 100)))
```

```python
for i in range(large_number):
    ...
```

```python
for i in xrange(large_number):
    ...
```

```python
class A(object):
    pass
```

```python
class A(object):
    __slots__ = ['a', 'b', 'c']
```

Conclusion

- PyPy is fast
- mature
- stable
- abstractions for free!
- (I wonder why you all are still here instead of busy trying PyPy :-))
    - not all C extensions are supported (numpy anyone?)
    - too much memory (sometimes)

Conclusion

- PyPy is fast
- mature
- stable
- abstractions for free!
- (I wonder why you all are still here instead of busy trying PyPy :-))
    - not all C extensions are supported (numpy anyone?)
    - too much memory (sometimes)

Contacts, Q/A

- `http://pypy.org`
- blog: `http://morepypy.blogspot.com`
- mailing list: pypy-dev (at) python.org
- IRC: #pypy on freenode
- `http://antocuni.eu`

?

Training session

- Run your application under PyPy

How to run PyPy

- `pypy program.py`
- That’s it!
  - (modulo details)

Challenge

- `html_fibo.py`
- HTML list of fibonacci numbers
- (the most complicate ever)
- run it on CPython
- run it on PyPy
- fix it!
- `http://pypy.org`
- `http://antocuni.eu/misc/html_fibo.txt`

Refcounting vs generational GC (1)

gc0.py

```
def foo():
    f = file('/tmp/bar.txt', 'w')
    f.write('hello world')

foo()
print file('/tmp/bar.txt').read()
```

gc1.py

```
def foo():
    f = file('/tmp/bar.txt', 'w')
    f.write('hello world')
    f.close() # <-------
```

gc2.py

```
def foo():
    with file('/tmp/bar.txt', 'w') as f:
        f.write('hello world')
```

Refcounting vs generational GC (1)

gc0.py

```
def foo():
    f = file('/tmp/bar.txt', 'w')
    f.write('hello world')

foo()
print file('/tmp/bar.txt').read()
```

gc1.py

```
def foo():
    f = file('/tmp/bar.txt', 'w')
    f.write('hello world')
    f.close() # <-------
```

gc2.py

```
def foo():
    with file('/tmp/bar.txt', 'w') as f:
        f.write('hello world')
```

Refcounting vs generational GC (1)

gc0.py

```
def foo():
    f = file('/tmp/bar.txt', 'w')
    f.write('hello world')

foo()
print file('/tmp/bar.txt').read()
```

gc1.py

```
def foo():
    f = file('/tmp/bar.txt', 'w')
    f.write('hello world')
    f.close() # <-------
```

gc2.py

```
def foo():
    with file('/tmp/bar.txt', 'w') as f:
        f.write('hello world')
```

Refcounting vs generational GC (2)

- `__del__`
  - especially files or sockets
  - don't leak file descriptors!
- weakrefs
- `finally` inside generators