

Introduction to PyPy

Antonio Cuni

BigDive 2012

October 22 2012



About me

- PyPy core dev
- PyPy py3k tech leader
- `pdb++`, `fancycompleter`, ...
- Consultant, trainer
- `http://antocuni.eu`
- twitter: `@antocuni`

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

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

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

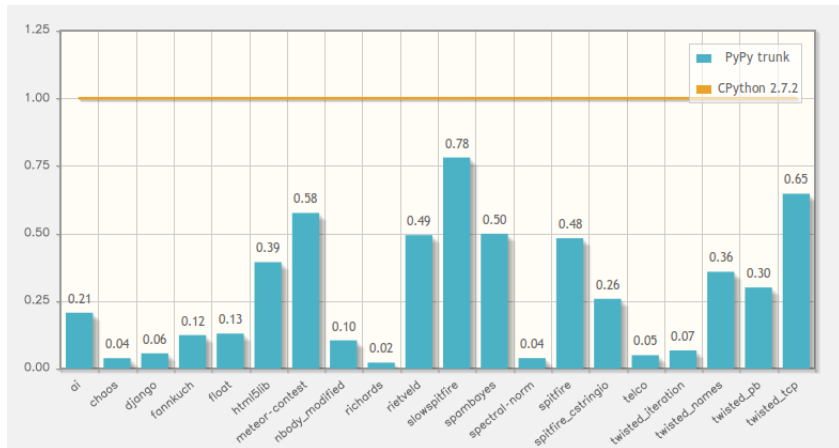
PyPy: Software archeology

- 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**
 - ▶ numpy (more on that later)

Speed



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__`)

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')
```

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')
```

Obscure details that people rely on

“There is No Feature Obscure Enough for people not to rely on”

- Non-string keys in `__dict__` of types
- Exact naming of a list comprehension variable
- Relying on untested and undocumented private stuff
- Exact message matching in exception catching code
- Refcounting details

Obscure details that people rely on

“There is No Feature Obscure Enough for people not to rely on”

- Non-string keys in `__dict__` of types
- Exact naming of a list comprehension variable
- Relying on untested and undocumented private stuff
- Exact message matching in exception catching code
- Refcounting details

import numpy (1)

● <http://buildbot.pyipy.org/numpy-status/latest.html>

	PyPy	PyPy	PyPy	PyPy	PyPy
ALLOW_THREADS	* dtype.__lt__	* generic.__xor__	* min	✓ negative	✓
BUFSIZE	* dtype.__mul__	* generic.all	* minimum	✓ newaxis	✓
CLIP	* dtype.__ne__	✓ generic.any	* mintypecode	* newbuffer	*
DataSource	* dtype.__new__	✓ generic.argmax	* mirr	* nonzero	✓
ERR_CALL	* dtype.__reduce__	✓ generic.argmin	* mod	* not_equal	✓
ERR_DEFAULT	* dtype.__reduce_ex__	✓ generic.argsort	* modf	* nper	*
ERR_DEFAULT2	* dtype.__repr__	✓ generic.astype	* msort	* npv	*
ERR_IGNORE	* dtype.__rmul__	* generic.base	* multiply	✓ number	✓
ERR_LOG	* dtype.__setattr__	✓ generic.byteswap	* nan	✓ obj2sctype	*
ERR_PRINT	* dtype.__setstate__	* generic.choose	* nan_to_num	* object	*
ERR_RAISE	* dtype.__sizeof__	* generic.clip	* nanargmax	* object0	*
ERR_WARN	* dtype.__str__	✓ generic.compress	* nanargmin	* object__	*
FLOATING_POINT_SUPPORT	* dtype.__subclasshook__	✓ generic.conj	* nanmax	* ogrid	*
FPE_DIVIDEBYZERO	* dtype.alignment	✓ generic.conjugate	* nanmin	* ones	✓
FPE_INVALID	* dtype.base	* generic.copy	* nansum	* ones_like	*
FPE_OVERFLOW	* dtype.byteorder	✓ generic.cumprod	* nbytes	* outer	*
FPE_UNDERFLOW	* dtype.char	✓ generic.cumsum	* ndarray	✓ packbits	✓
False_	✓ dtype.descr	* generic.data	* ndarray.T	✓ pi	✓
Inf	✓ dtype.fields	✓ generic.diagonal	* ndarray.__abs__	✓ piecewise	*
Infinity	✓ dtype.flags	* generic.dtype	* ndarray.__add__	✓ pkgload	*
MAXDIMS	* dtype.hasobject	* generic.dump	* ndarray.__and__	* place	*
MachAr	* dtype.isbuiltin	* generic.dumps	* ndarray.__array__	* pmt	*
NAN	✓ dtype.isnative	* generic.fill	* ndarray.__array_finalize__	* poly	*
NINF	* dtype.itemsize	✓ generic.flags	* ndarray.__array_interface__	✓ polyld	*
NZERO	✓ dtype.kind	✓ generic.flat	* ndarray.__array_priority__	* polyadd	*
NaN	* dtype.name	✓ generic.flatten	* ndarray.__array_struct__	* polyder	*
PNF	✓ dtype.names	✓ generic.getfield	* ndarray.__array_wrap__	* polydiv	*
PZERO	✓ dtype.newbyteorder	* generic.imag	* ndarray.__class__	✓ polyfit	*

import numpy (2)

- in-progress, funded by donations
- JIT-friendly
- almost as fast as the corresponding C code
- be happy with pure Python loops
- the bad news: scipy not there (yet)

numpy quick benchmarks

numpybench.py

```
def c_loop(a):  
    return numpy.sum(a)  
  
def pyloop(a):  
    sum = 0  
    for i in range(len(a)):  
        sum += a[i]  
    return sum
```

numpybench2.py

```
def c_loop(a, b, c):  
    return numpy.add(a, numpy.multiply(b, c))  
  
def pyloop(a, b, c):  
    N = len(a)  
    assert N == len(b) == len(c)  
    res = numpy.zeros(N)  
    for i in range(N):  
        res[i] = a[i] + b[i]*c[i]  
    return res
```

numpy quick benchmarks

numpybench.py

```
def c_loop(a):  
    return numpy.sum(a)  
  
def pyloop(a):  
    sum = 0  
    for i in range(len(a)):  
        sum += a[i]  
    return sum
```

numpybench2.py

```
def c_loop(a, b, c):  
    return numpy.add(a, numpy.multiply(b, c))  
  
def pyloop(a, b, c):  
    N = len(a)  
    assert N == len(b) == len(c)  
    res = numpy.zeros(N)  
    for i in range(N):  
        res[i] = a[i] + b[i]*c[i]  
    return res
```

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 metaprogramming 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 sobelidx(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

```
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);
}
```

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

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

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

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

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

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

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

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

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

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

```
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)
```

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

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

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

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

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

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

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

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

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

Pointless optimization techniques

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

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

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

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

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

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

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

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

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

```
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']
```

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