Alternate Pythons

Boston Python Meetup, 2011-08-17

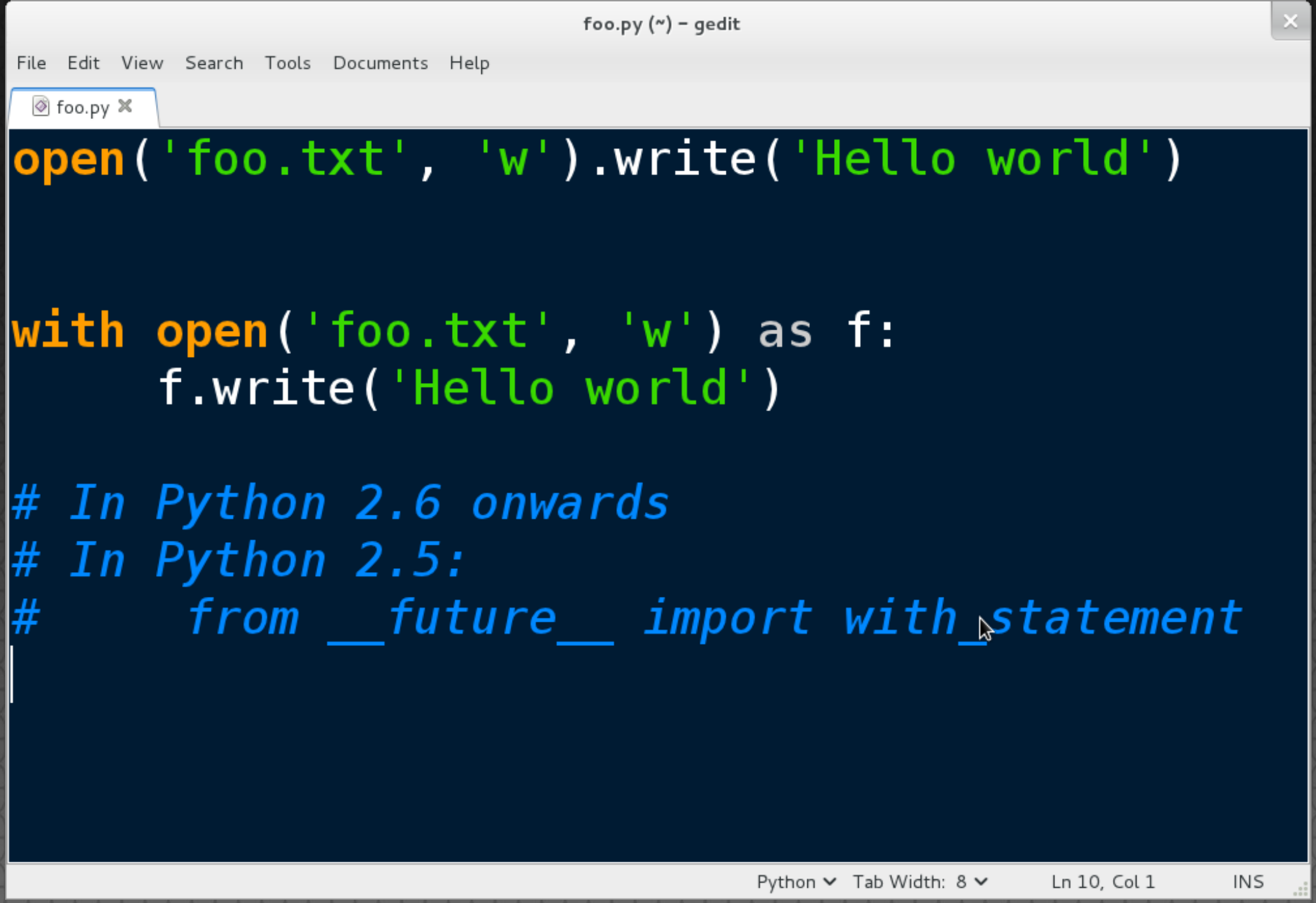
Presented by
David Malcolm
<dmalcolm@redhat.com>

Overview

1. “Python” vs “CPython”
2. Inside a Python implementation
3. Jython and IronPython
4. PyPy
5. Other runtimes
6. Which Python is right for you?

Informal survey

“Python” vs “CPython”


foo.py (~) – gedit
File Edit View Search Tools Documents Help
foo.py
open('foo.txt', 'w').write('Hello world')
with open('foo.txt', 'w') as f:
f.write('Hello world')
# In Python 2.6 onwards
# In Python 2.5:
# from __future__ import with_statement
Python Tab Width: 8 Ln 10, Col 1 INS

Inside a Python runtime

- A type system: objects and types
- Memory management
- Parser
- Execution (e.g. an interpreter)
- Standard library
- API for extensions

CPython

CPython

- The original Python implementation.

  First released in 1991, hacked on by a cast of hundreds

- The runtime (the type system, parser and interpreter, API for extensions)
  - about 200,000 lines of .c code
- The standard library:
  - about 200,000 lines of .c code
  - about 400,000 lines of .py code
- Thousands of add-on packages:

  http://pypi.python.org/pypi has about 12000 packages (mostly in .py, with some .c)

fedora

CPython's object system

Object-handling

- C implementation of a type system, with the usual menagerie of objects and types, all hand-coded in C:
  - lists, tuples, dictionaries, floats, strings, integers (both word-sized and arbitrary-sized), user-definable types, etc
- Objects are .c structures, with a reference count
  - One giant mutex: the “global interpreter lock”
- References between objects are just .c pointers
  - Objects can't move around in memory
- Lots of fiddly reference-counting, done “by hand”
  - Easy to get wrong
  - Supposedly read-only operations are modifying the refcounts

CPython's interpreter

A simple stack-based virtual machine, with its own bytecode format

- .py files are parsed into syntax trees
- syntax trees are converted into CPython bytecode
- simple optimization pass
  - many interesting optimizations are not possible due to the language being too dynamic
- the result is cached as a .pyc file; usually we can skip all of the above

CPython's interpreter

The opcode dispatch loop

(massively simplified! can you spot the recounting bug below?)

```
1       while (notdone) {
2         switch (next_opcode) {
3
4           case NOP:
5             /* do nothing */
6             break;
7
8           case BINARY_ADD:
9             w = POP();
10            v = TOP();
11            x = PyNumber_Add(v, w);
12            Py_DECREF(v);
13            Py_DECREF(w);
14            SET_TOP(x);
15            break;
16
17          /* etc, about 150 opcodes */
18
```

CPython's libraries

- lowest level of standard library written in .c
- much of standard library written in .py

CPython: the good parts

- It's very easy to write extension modules that glue CPython together with C code.

  (Perhaps too easy! Please stop crashing /usr/bin/python with your bugs)
- It's relatively easy to embed CPython into an application for use as a scripting language, exposing that application's native types in a "pythonic" way. (e.g. httpd, gdb, gedit, rhythmbox)
- The insides are fairly simple (if you know C): relatively easy to debug

What's ...not so good?

Too slow?

- Bytecode dispatch is never going to be as fast as machine code
- Very dynamic: lots of jumping through function pointers, with little possibility of type-based optimizations

Global Interpreter Lock ("GIL")

- One big mutex surrounding all Python object usage, including the bytecode interpreter loop.
- Keeps things simple, but blocks much parallelism

Reference-counting “fun”

- It's too easy to get this wrong and leak memory (or cause a crash)
- Objects can't move around in memory: can fragment the heap
- Lots of refcount twiddling: impossible to have readonly data in shared memory pages (e.g. KVM's KSM)

Non-opaque object API

The implementation details of the objects are visible to C extensions.

- This makes them hard to change without breaking hundreds of extensions
  - So e.g. strings are merely length + buffers of bytes, rather than e.g. ropes

Debug Builds

CPython: --with-pydebug

(and various other configure-time options)

- Adds lots of useful debugging instrumentation
- Easier to debug in gdb
- Halves the speed of the resulting binary (roughly)
- .pyc/.pyo files are compatible with the regular optimized python
- .so files are **not compatible** with the regular optimized python (ABI differences)

fedora

Jython

Jython

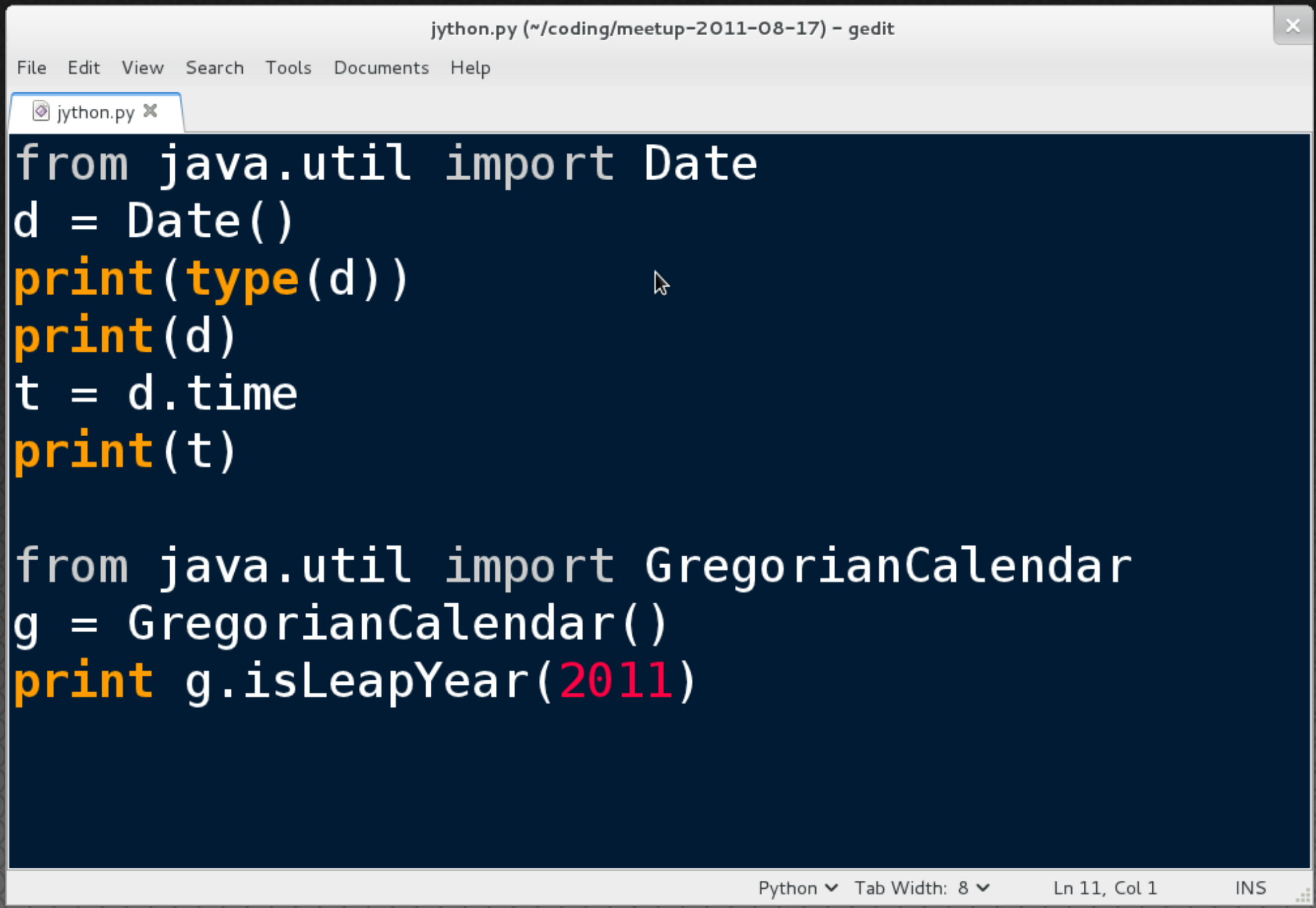
- Implementation of Python in Java, on top of the JVM
  - About 90k lines of Java code, some of it autogenerated

Jython: Objects

Java base class:

org.python.core.PyObject

- Analagous menagerie of subclasses e.g. PyDictionary etc, implementing Python object/type semantics e.g. __call__, __iter__
- Can wrap arbitrary Java objects
- Python standard library still available


jython.py (~/coding/meetup-2011-08-17) - gedit
File Edit View Search Tools Documents Help
jython.py
from java.util import Date
d = Date()
print(type(d))
print(d)
t = d.time
print(t)
from java.util import GregorianCalendar
g = GregorianCalendar()
print g.isLeapYear(2011)
Python
Tab Width: 8
Ln 11, Col 1
INS

fedora

Jython: the runtime

- It's java
- The .py files are compiled to syntax trees, then converted directly into Java bytecode
- In theory, ought to be fast (JIT-compiled machine code)
  - However, much of the time the Java bytecode is calling back into the org.python.core.PyObject code, which has to implement some messy switch statements
    - (e.g. __call__ with variable-length arguments)

Jython: Advantages

- ability to work with pre-existing Java code
- embeddable inside a Java appserver (comes as a .jar file)
- the Java VM has:
  - JIT compilation
  - garbage collection
  - years of research and aggressive competition amongst JVM vendors
  - no GIL

Jython: Disadvantages

- No C extensions, to my knowledge
- Not readily embeddable in .c code
- Still at python 2.5

IronPython

IronPython

- Similar to Jython, but in C#, on top of the .Net runtime
- Works on Linux with Mono
- Some ability to run CPython extensions via a project named "Ironclad"

PyPy

PyPy

- Radically different approach to the above
- Implementation of an interpreter for the full Python language (with JIT compilation)
- Written in a high-level language
- Many of the implementation details are "pluggable" policies:
  - implementations of the python objects
  - memory allocation/garbage collection
  - GIL

PyPy

- The implementation language is then compiled down to .c code, from which we obtain a binary

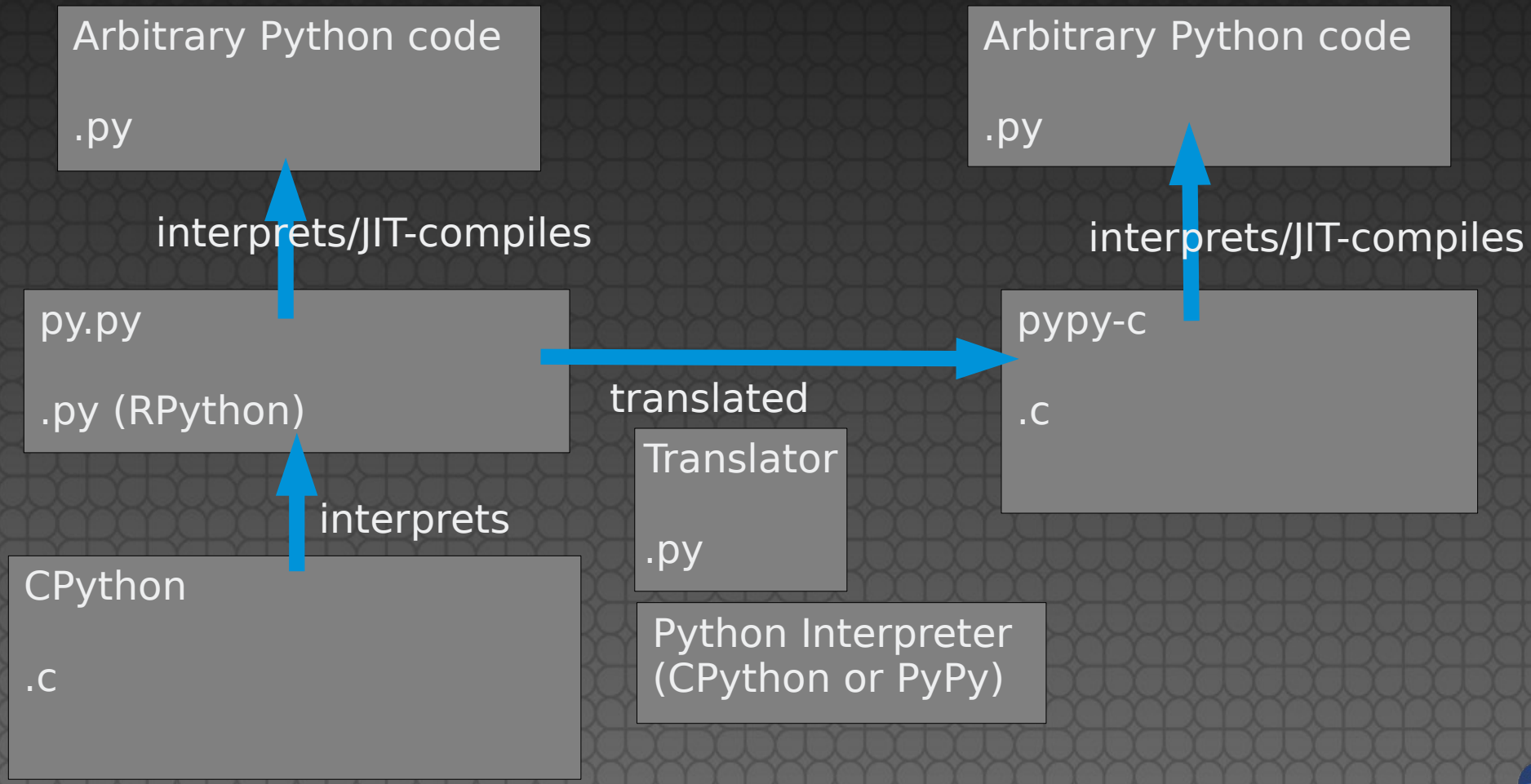
  (It can also be compiled to Java and to C#, but these backends are much less mature)

This allows us to build a near-arbitrary collection of different implementations of Python with differing tradeoffs

PyPy

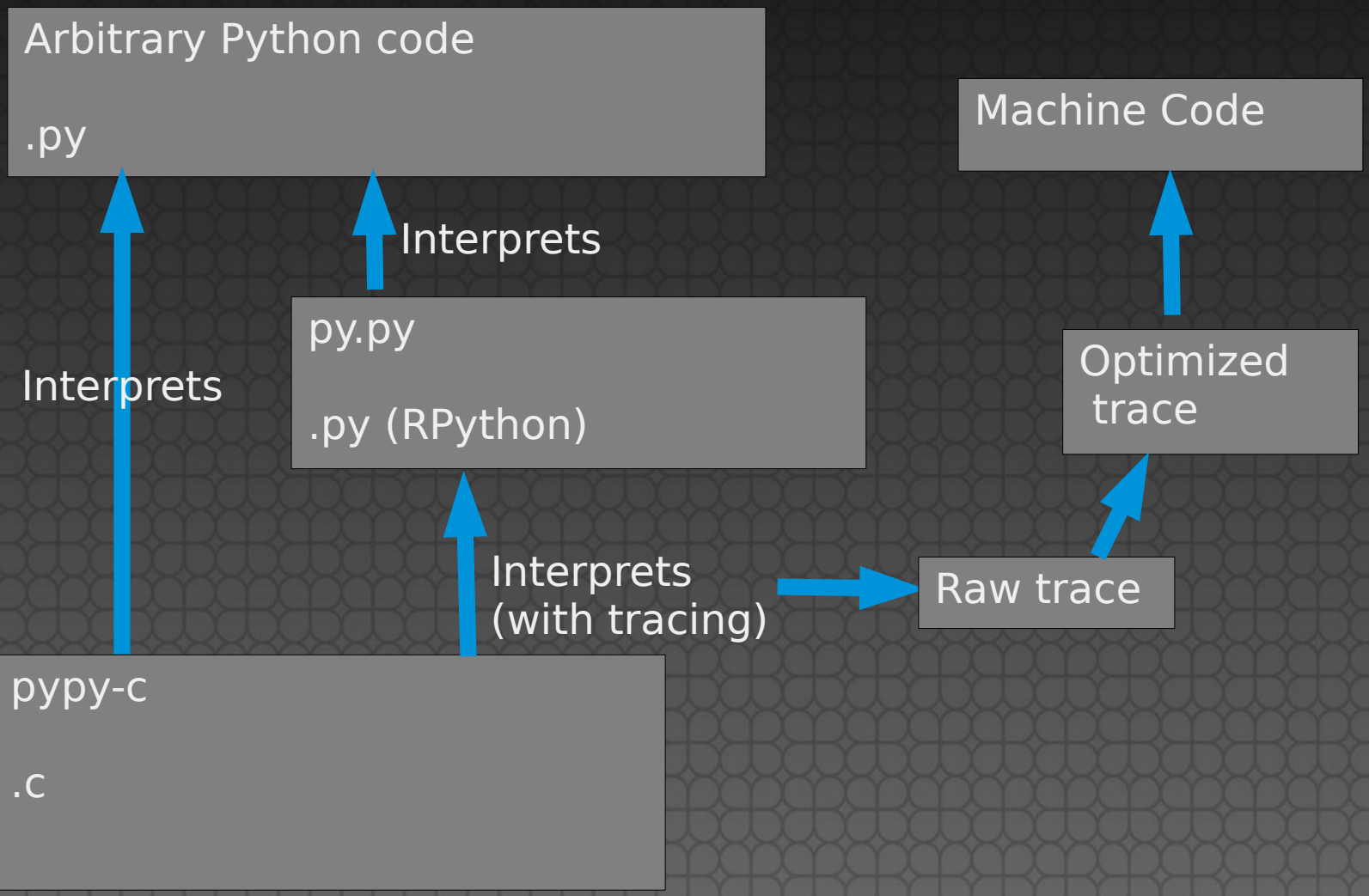
- The implementation language is Python
  - albeit a restricted subset "RPython" (chosen to making it amenable to being compiled)
- The result implements the full Python language though
- Can also be run without translation, directly on another Python runtime (about a thousand times slower)
  - e.g. on top of CPython, or on top of itself

How many Pythons?



fedora

JIT-compilation



fedora

PyPy

- PyPy does have .pyc files by default (it's configurable, of course)
  - They are similar to, but different from, CPython 2.7's .pyc files (different magic number)
- Beginning to have support for extension modules
  - Same API as CPython, in theory
  - Different ABI (i.e. separate recompilation needed)
  - I haven't managed to get this to work yet

Advantages of PyPy

- Speed: see http://speed.pypy.org
  - Better object implementations (smarter data structures)
  - Just-In-Time compilation, based on tracing itself interpreting “hot” loops
- Memory usage
  - Ought to be smaller: better data structures

Disadvantages of PyPy

- 6 million lines of autogenerated .c code.

  (I'm working on patches to improve the readability of this code)

  - “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it” Brian Kernighan
- C extension support not yet mature
- Has its own machine code generator (for the JIT)
- Build system also postprocesses assembler output from GCC

Other Implementations?

Other implementations

- Cython
- Unladen Swallow
- Psyco
- Pynie
  - Implementation on top of Parrot VM
- GCC Frontend

Which Python is right for you?

Tradeoffs

- Speed
- Memory usage
- “Debuggability”
- Interactions with other technologies

fedora™

What are you using Python for?

As a scripting language...

- One-off scripts
- Simple hacks that can can be changed into something more long-term
- A highly-readable high-level language

  (Alas, this is not a tautology)
- “Batteries included”

fedora™

Extensions

...as glue code for bridging low-level libraries with high-level code

- Easy to integrate with C and C++ code
- Relatively easy to debug the result when the low-level library crashes
- As an embedded scripting language
- In Fedora/RHEL, most of our system tools are written in Python (e.g. Installer, upgrader, etc)

Web development

Dynamically generating HTML (and JSON, XML, etc)

- Django
- TurboGears, Pyramid, etc

Other uses?

fedora™

Questions & Discussion

Contact:
dmalcolm@redhat.com

Python 3

Python 3

Big rewrite of CPython 2, fixing lots of long-standing warts

- Differences in syntax to Python 2
- Changes to standard library
- Different .pyc files
- Much nicer to use than Python 2 (IMHO)
- Slowly growing 3rd-party module support
- On schedule: give it a year or two

Packaging concerns

- PEP 3147: parallel-management of .pyc files
- PEP 3149: parallel-installable .so files

src.rpm	Runtime	Python Version	.pyc files	.so files	Notes
python	CPython 2 (optimized)	2.7	Yes	Yes	
	(debug)	2.7	Yes	Yes	
python3	CPython 3 (optimized)	3.2	Yes	Yes	
	(debug)	3.2	Yes	Yes	
jython		2.5	No	No	
pypy		2.7	Yes	Yes	