Python 3.3:

Trust Me, It's Better Than Python 2.7

Dr. Brett Cannon
http://about.me/brettcannon

March 2013

Features of Python 3.0 through 3.3

Stuff you already have in (at least) Python 2.7

- Requires a `__future__` statement
  - Absolute import
  - Unicode literals
  - "New" division
  - `print` function
- Set literals
- Set and dict comprehensions
- Multiple context managers
- C-based `io` library
- `memoryview`
  - New implementation in Python 3.3
- `future_builtins`
- `except Exception as exc: ...`
- `str.format()`
  - Python 2.7 added auto-numbering (e.g. `'{} {}'.format(0, 1)`)
- `numbers`

Minor features (that don't need an entire slide to explain)

- Dict views
- Comparison against disparate types is a TypeError
    - `2 > 'a'`
- `raise Exception, 42` is a no-no
- Metaclasses
    - `metaclass` class argument
    - `__prepare__`
- Standard library cleanup
- `argparse`
- Dictionary-based configuration for logging
- wsgi 1.0.1
- `super()`
- Unified integers
- `__next__()`

Minor features added in Python 3.3

- Reworking the OS and I/O exception hierarchy
- New modules
  - `lzma`
  - `ipaddress`
  - `faulthandler`
- `email` package's new policy & header API
- Key-sharing dictionaries
  - OO code can save 10% - 20% on memory w/o performance degradation

Features fancy/complicated enough to need their own slides

nonlocal

```
>>> def scoping():
...   higher_scoped = 0
...   def increment():
...     nonlocal higher_scoped
...     higher_scoped += 1
...   def value():
...     return higher_scoped
...   return inc, value
...
>>> increment, value = scoping()
>>> increment(); increment()  # 0 + 1 + 1 = 2
>>> value()
2
```

Extended iterable unpacking

```
>>> first, *rest = range(5)
>>> first
0
>>> rest
[1, 2, 3, 4]


>>> a, *b, c = range(5)
>>> a
0
>>> b
[1, 2, 3]
>>> c
4
```

Stable ABI

- Defines a "stable set of API functions"
- "Guaranteed to be available for the lifetime of Python 3"
- "Binary-compatible across versions"

concurrent.futures

```
def big_calculation(num):
    return num ** 1000000

arguments = list(range(20))

# Takes 6 seconds ...
list(map(big_calculation, arguments))

# Takes 1 second ...
from concurrent import futures
with futures.ProcessPoolExecutor() as executor:
    list(executor.map(big_calculation, arguments))
```

`decimal` module implemented in C

- New in Python 3.3
- Preview of benchmark results: 30x faster than pure Python version not uncommon (seen as high as 80x)!

Qualified names

- New in Python 3.3
- `__name__ + '.' + self.__qualname__` should give you the fully qualified name for an object now

```
>>> class C:
...   def f(): pass
...
>>> C.f.__name__
'f'
>>> C.f.__qualname__
'C.f'
```

yield from

- New in Python 3.3
- Allows generators to be factored out and replaced with a single expression
    - "`yield from` is to generators as calls are to functions"

```
def stupid_example():
    yield 0; yield 1; yield 2


def factored_out():
    yield 1; yield 2

def refactored_stupid():
    yield 0
    yield from factored_out()
```

`venv`

- New in Python 3.3
- Essentially `virtualenv` redone as part of Python itself
  - Creates an isolated directory where the system/user-wide site-packages directory is ignored
- Ways to create a virtual environment
  - `python3 -m venv /path/to/new/virtual/environment`
  - `pyvenv /path/to/new/virtual/environment`
- "a Python virtual environment in its simplest form would consist of nothing more than a **copy or symlink** of the Python binary accompanied by a `pyvenv.cfg` **file** and a site-packages **directory**"

BIGGER features/themes that need lots of slides

Exceptions

Included traceback

```
>>> import traceback
>>> try:
...   raise Exception
... except Exception as exc:
...   traceback.print_tb(exc.__traceback__)
...
  File "<stdin>", line 2, in <module>
```

Implicit exception chaining

```
>>> try:
...     raise Exception
... except Exception:
...     raise NotImplementedError  # __context__ set
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception

During handling of the above exception, another
exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotImplementedError
```

Explicit exception chaining

```
>>> try:
...     raise Exception
... except Exception as exc:
...     # Sets __cause__
...     # In Python 3.3, ``from None`` suppresses
...     raise NotImplementedError from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotImplementedError
```

Import

`importlib` as import (see my other talk for details)

- New in Python 3.3
- Pure Python implementation of import
  - All VMs *should* end up using the same implementation of import
- Allows for easier customization
- Easier writing of importers
- Logic of import, at a high level, is *much* simpler

Finer-grained import lock

- New in Python 3.3
- Importing in a thread used to cause deadlock
    - Could be unintended when a thread called a function that had a local import
        - E.g. functions in `os --` in order to allow for faster startup -- were often a trigger by doing local imports
- Now threads block until the import completes
    - If deadlock possible (read: circular import), then partially initialized modules allowed

`__pycache__` directories

- All `.pyc` files now kept in a __pycache__ directory
- `.pyc` filenames contain the interpreter and version number
  - Allows for different interpreters and versions of Python to have `.pyc` files without overwriting each other
- You can still distribute only `.pyc` files without source
  - ... unfortunately

Namespace packages

- New in Python 3.3
- All directories with **no** `__init__.py` file but whose name matches that of the package being imported are collected and set as the `__path__` for an empty module
- Namespace modules set typical attributes **but** `__file__`
- Any change to the `__path__` of the parent package (or `sys.path` when no parent) triggers a recalculation of `__path__` for the namespace package
    - E.g. if `monty.__path__` changes, then `monty.python.__path__` is recalculated on access
- All previous imports (i.e. regular packages, modules) continue to work and take precedence over namespace packages

Functions

Keyword-only arguments

- Great for expanding pre-existing APIs
    - Never have to worry about a programmer accidentally using a new API by passing more arguments than expected

```
def monty_python(bacon, spam, *, different=None):
  pass
```

Function annotations

- Can annotate any parameter and the return value with any object
  - Does not have to be type-specific!
  - Standard library explicitly does not use function annotations to allow community to decide how to use them

```
spam = None
bacon = 42

def monty_python(a:spam, b:bacon) -> "different":
  pass
```

Function signature objects

- New in Python 3.3
- Provides an object representation of every detail of a callable's signature
  - Names
  - Annotations
  - Default values
  - Positional, keyword (only)
- Can use to calculate how arguments would be bound by a call
- Can create objects from scratch, allowing for adding parameter details to callables that typically don't have such details
  - E.g. C-based functions

Unicode, unicode, unicode!

Unicode while you code!

- UTF-8 is the default encoding for source code
- Non-ASCII identifiers
  - Not *everything* in the entire Unicode standard, but a lot is allowed

Unicode while specifying string literals!

- All string literals are Unicode
    - The `u` prefix is allowed in Python 3.3 and is a no-op
        - Allows for specifying bytes, unicode, and native strings in Python 2.7 vs 3.3 syntactically
        - `'native string'`
            - Used when you work with ASCII text **only**
            - Python 2.7: `str` type
            - Python 3.3: `str` type
        - `u'always Unicode'`
            - Python 2.7: `unicode` type
            - Python 3.3: `str` type
        - `b'some bytes'`
            - Python 2.7: `bytes` type (alias for `str`)
            - Python 3.3: `bytes` type
    - `from __future__ import unicode_literals`
- Biggest porting hurdle when you have not clearly delineated what is text vs. what are bytes

Better Unicode during execution in Python 3.3!

- Python chooses the most compact representation for a string internally
  - Latin-1, UTF-16, or UTF-32
- No more narrow vs. wide builds!
  - Extensions will no longer need to be built twice
  - Python can now always represent any Unicode character unlike a narrow build with non-BMP characters
- Memory usage compared to Python 2.7
  - Narrow build smaller in Python 2.7 in a Django benchmark by less than 8%
  - **Python 3.3 smaller** compared to a wide build of Python 2.7 by more than 9%

Who's faster?

How I benchmarked

- Compiled from the same checkout on the same day
    - 2.7 and 3.3 branch
    - Decided not to download binaries as building from scratch was easier and you will all eventually be running that code anyway
    - UCS4/wide build for a more equal comparison of abilities
- Results are relative between the two binaries
    - Means low-level details don't really matter as they equally affect both binaries
- Results from a Core i7 MacBook Pro running OS X 10.8
- Used the unladen benchmarks + extras
    - http://hg.python.org/benchmarks
    - Now includes as many PyPy benchmarks as possible
    - Used some libraries which do not have released Python 3 support officially -- but have it in code repository -- so not entirely what is publicly available

If you sorted all of the benchmarks and looked at the median result ...

Python 3.3 is **THE SAME**

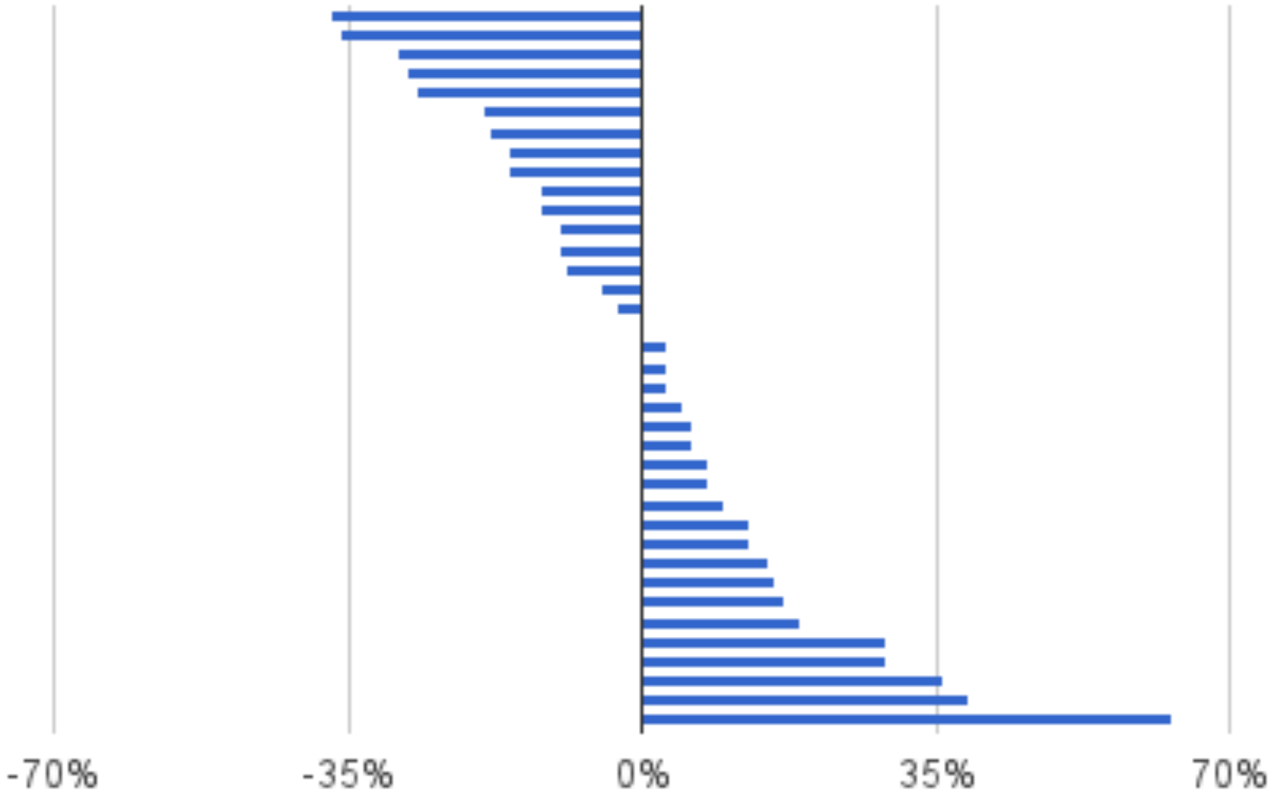
Worst benchmark result

startup_nosite is **0.73x slower**

Best benchmark result

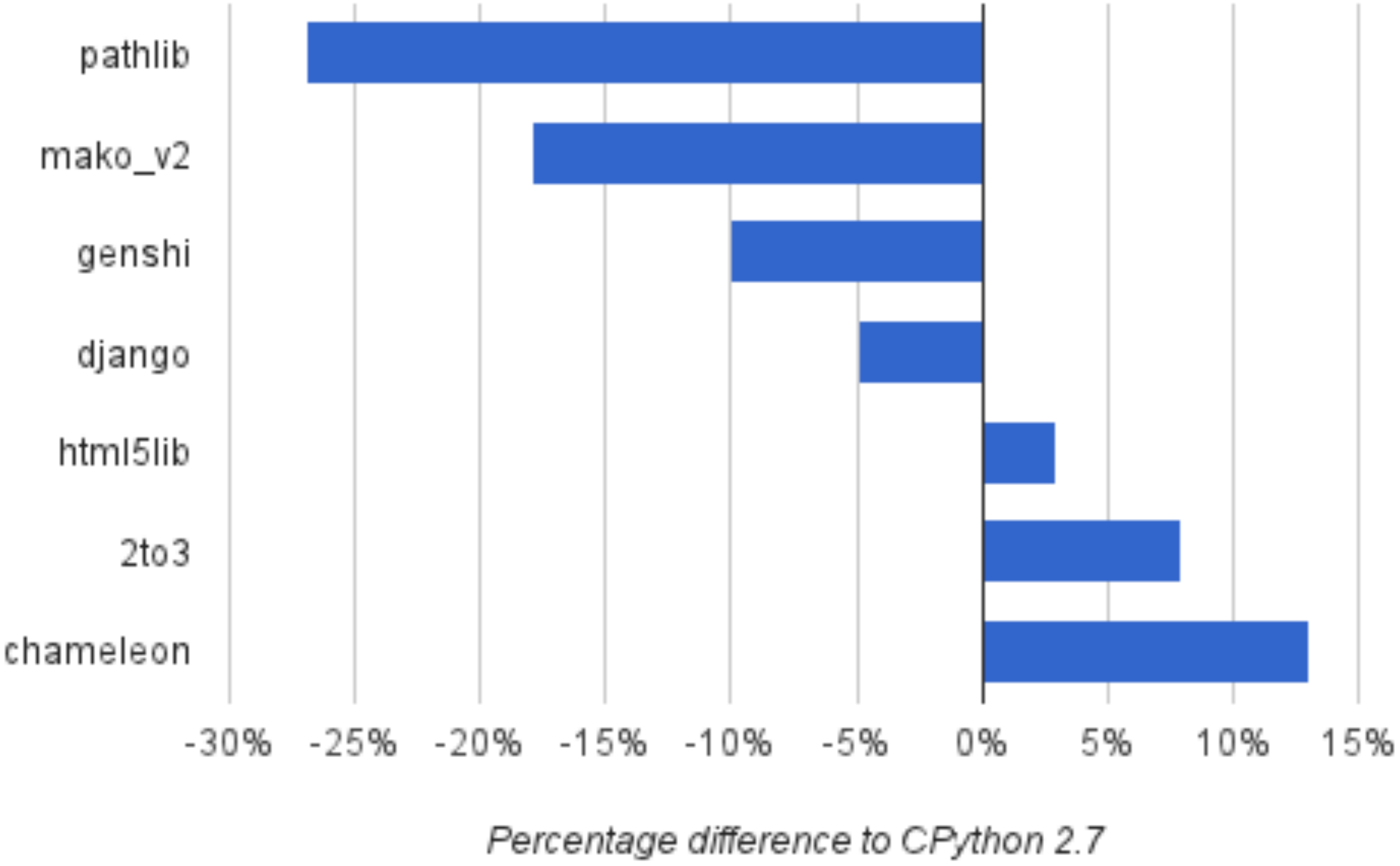
telco is **46.96x faster**

(Most) benchmark results



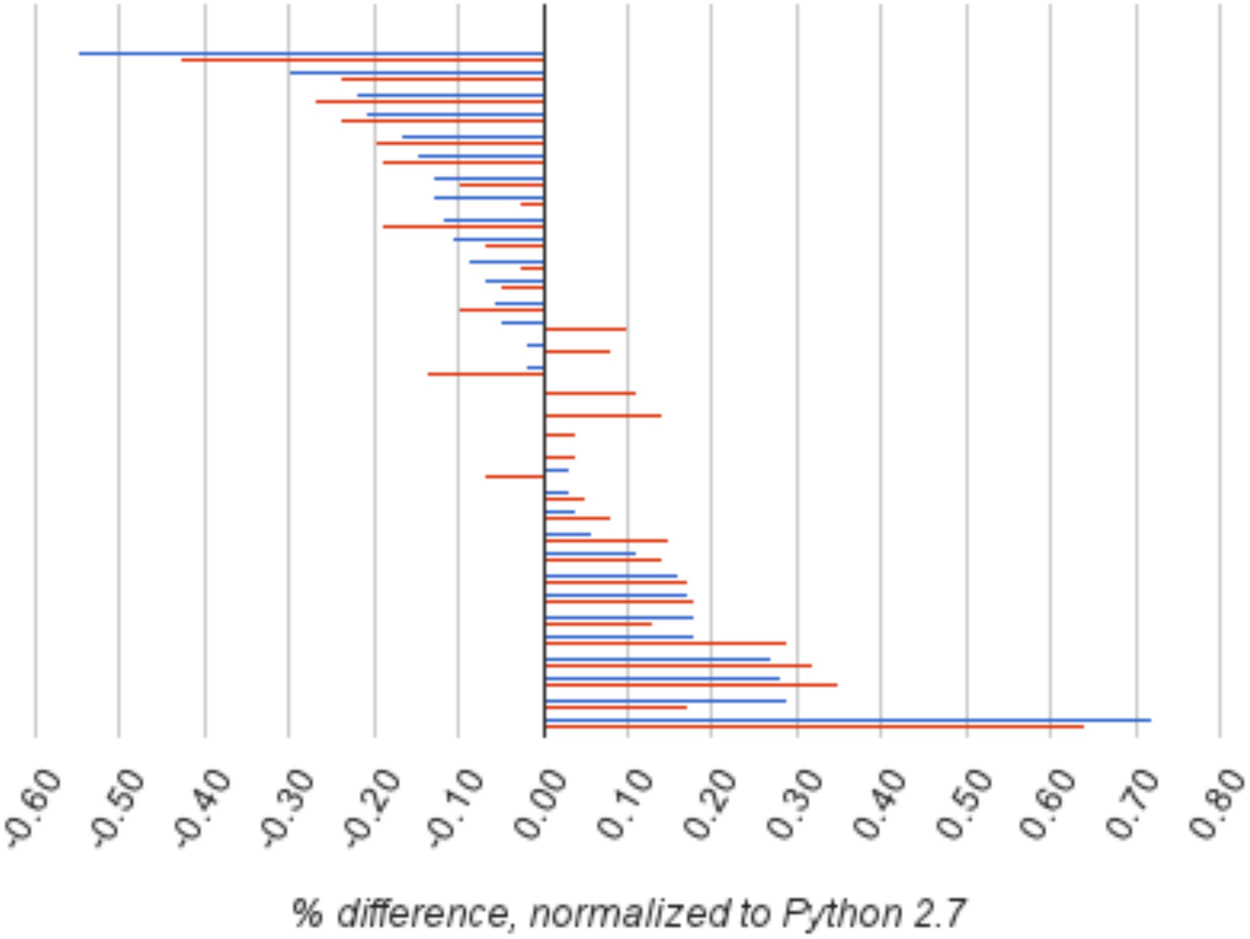
Percentage difference compared to CPython 2.7

Macro benchmarks

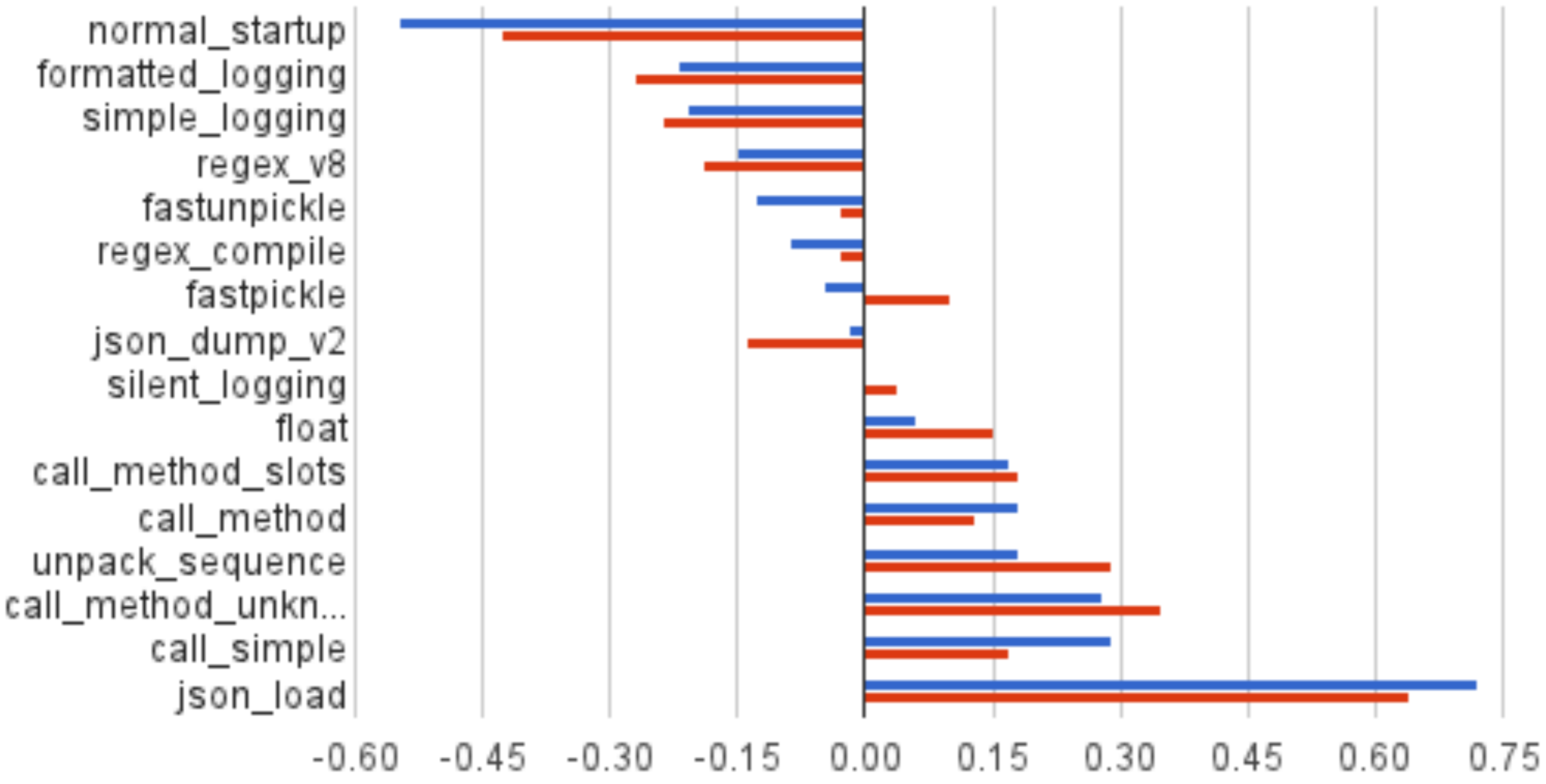


Q&A

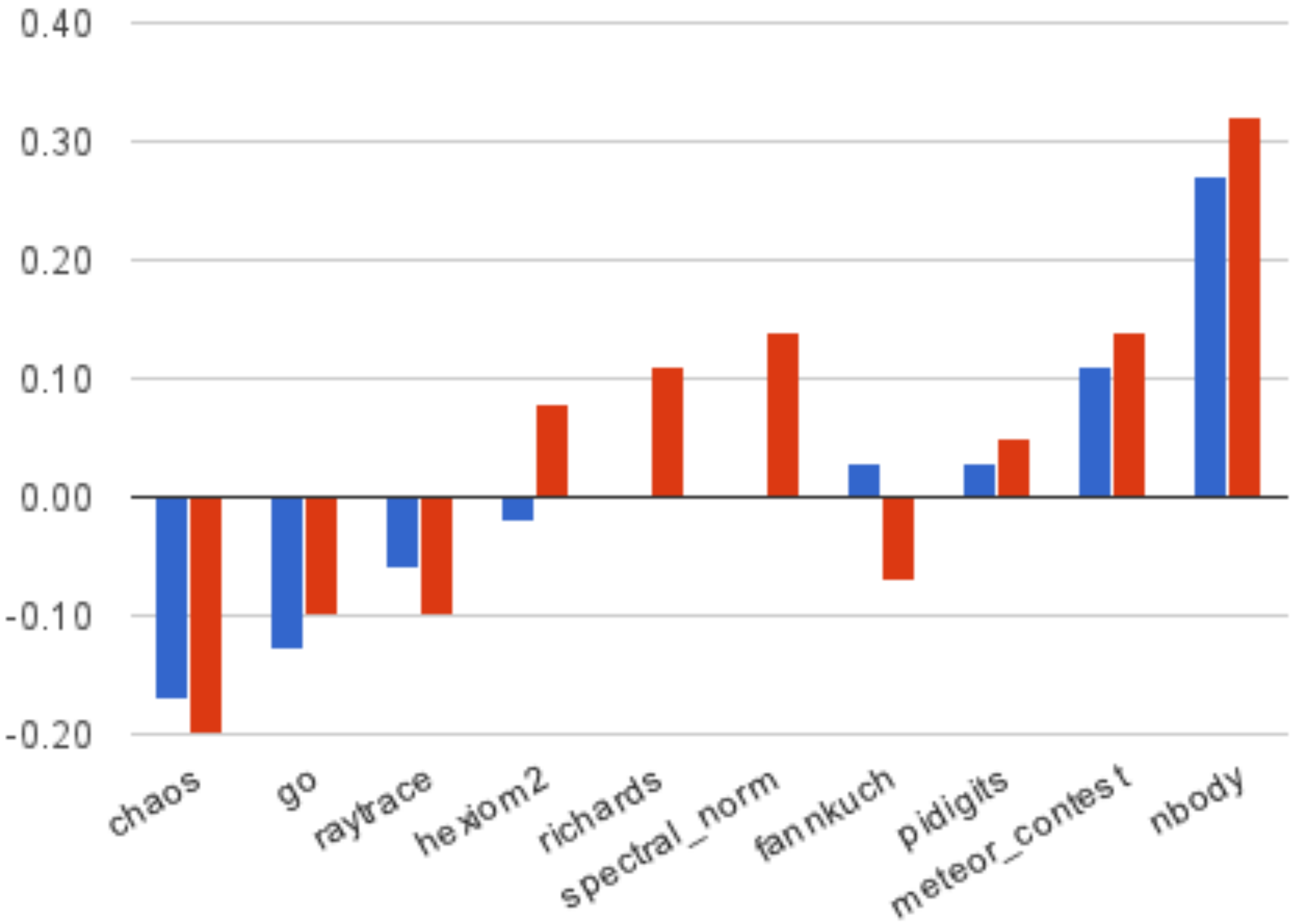
(Most) Benchmark results

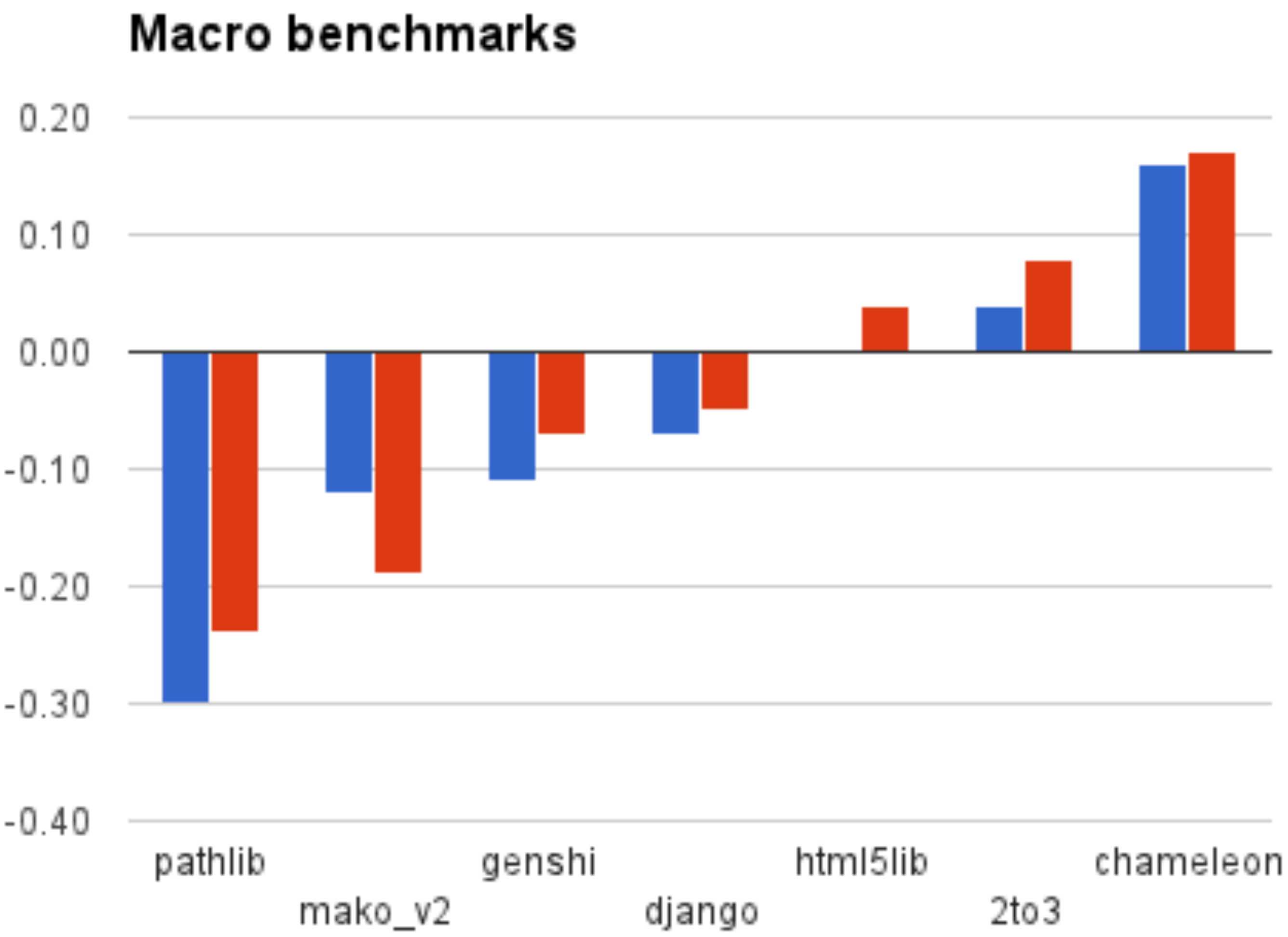


Synthetic benchmarks



Algorithmic benchmarks




Macro benchmarks
0.20
0.10
0.00
-0.10
-0.20
-0.30
-0.40
pathlib
mako_v2
genshi
django
html5lib
2to3
chameleon

Macro benchmark numbers

pathlib (0.6)	-0.30	-0.24
mako_v2 (0.7.3)	-0.12	-0.19
genshi (trunk)	-0.11	-0.07
django (1.5.0a1)	-0.07	-0.05
html5lib (trunk)	0.00	0.04
2to3 (2.6)	0.04	0.08
chameleon (2.9.2)	0.16	0.17