PYTHON IS SLOW

Make it faster with C

Ben Shaw

"It's OK that Python isn't fast, you can write your slow functions in C!"

–Everyone

TABLE OF CONTENTS

C Module vs C Types

TABLE OF CONTENTS

A Simple Algorithm

TABLE OF CONTENTS

5 Different Implementations

TABLE OF CONTENTS

Speed Comparisons

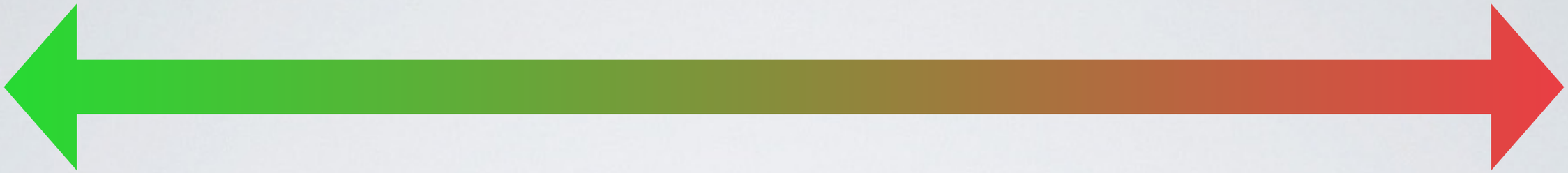
TABLE OF CONTENTS

- C Module vs C Types
- A Simple Algorithm
- 5 Different Implementations
- Speed Comparisons

PYTHON C INTEGRATION

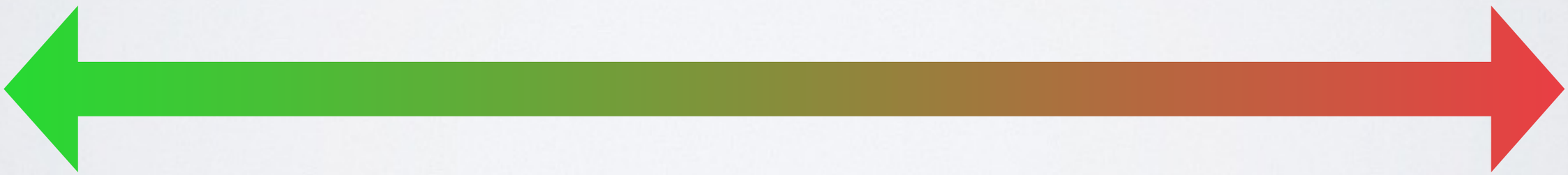
C Python Module	CTypes
• The more traditional way • Write special wrapper code in C • More Pythonic • Integrate exceptions, help text etc	• Newer (Only available since Python 2.5/2006 😓) • Simpler to implement, only need to write Python code

CODE CONTINUUM



Python CTypes C Module C

PYTHON INTEGRATION



Python C Module CTypes C

A SIMPLE BUT COMPUTATIONALLY EXPENSIVE EXAMPLE

IS A NUMBER PRIME?

SUPER NAÏVE

```
for divisor in xrange(√number):*
  if number % divisor == 0:
    return False
return True
```

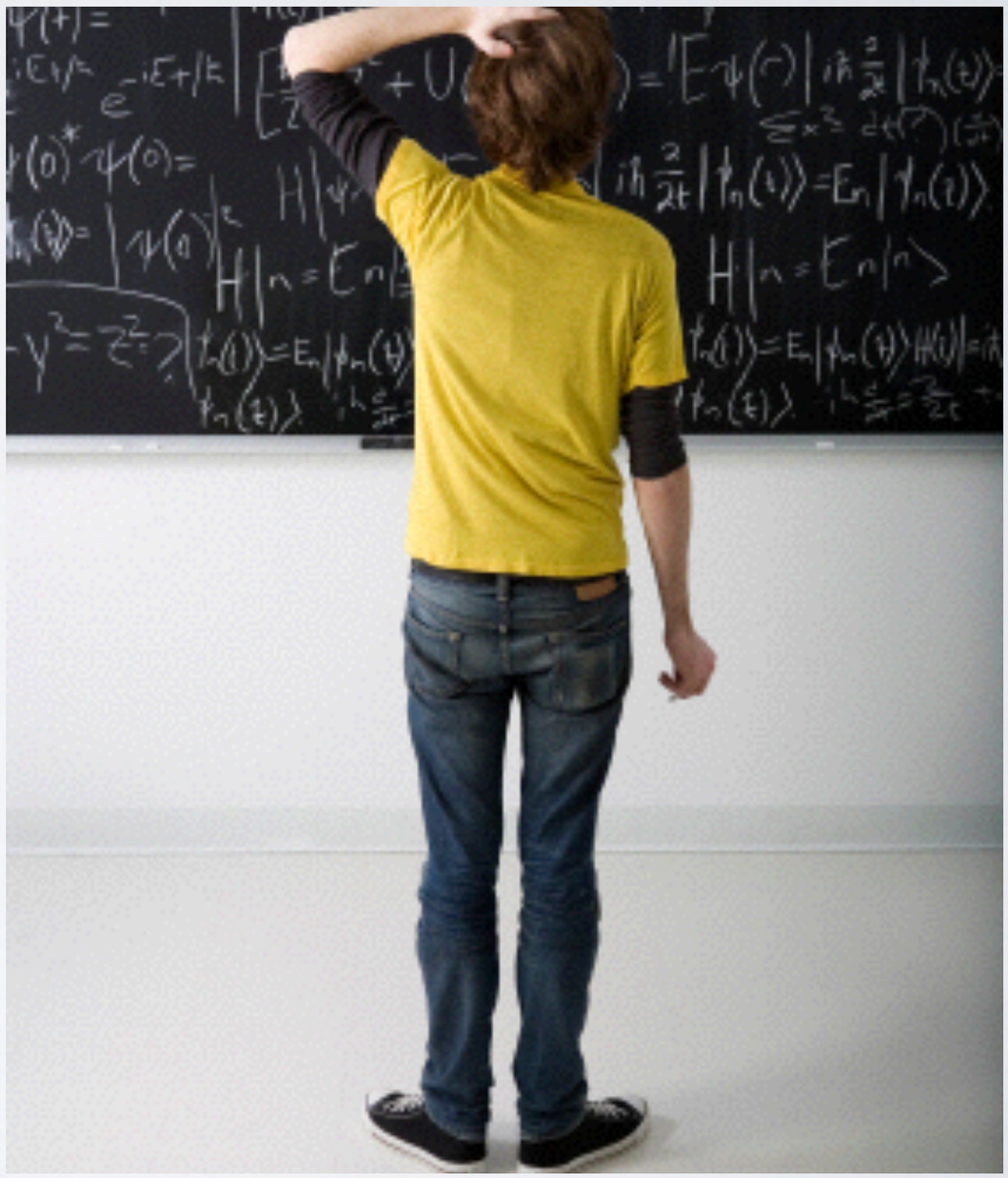
*With provisions for 1 and 2

ITERATE OVER A BUNCH OF NUMBERS

and do math with each one.

LOL REAL WORLD EXAMPLE

- Easy to understand
- Easy to implement
- Runs faster in C than in Python



PRIME 5 WAYS

- Pure Python (CPython)
- Pure C
- Python with C Module
- Python with C Types
- PyPy

PRIME 5 WAYS

- **Pure Python (CPython)**
- Pure C
- Python with C Module
- Python with C Types
- PyPy

PURE PYTHON

```
def is_prime_naive(number):
    # return false for even and 1, true for 2
    test_max = prime_test_max(number)

    for divisor in xrange(3, test_max + 1, 2):
        if number % divisor == 0:
            return False

    return True
```

PURE PYTHON

```
from pythonlib.con_math import is_prime_naive
from pythonlib.wrappers import main


if __name__ == '__main__':
    main(is_prime_naive)
```

PURE PYTHON

```
from pythonlib.con_math import is_prime_naive
from pythonlib.wrappers import main


if __name__ == '__main__':
    main(is_prime_naive)
```

PURE PYTHON

```
def main(prime_func):
    # skip some parsey stuff
    if argv[1] == '-b':
        benchmark_maximum = int(argv[2])
        benchmark(benchmark_maximum, prime_func)
    else:
        num_to_check = int(argv[1])

        if prime_func(num_to_check):
            print "{} is prime.".format(num_to_check)
        else:
            print "{} is not prime.".format(num_to_check)
```

PURE PYTHON

```
def main(prime_func):
    # skip some parsey stuff
    if argv[1] == '-b':
        benchmark_maximum = int(argv[2])
        benchmark(benchmark_maximum, prime_func)
    else:
        num_to_check = int(argv[1])

        if prime_func(num_to_check):
            print "{} is prime.".format(num_to_check)
        else:
            print "{} is not prime.".format(num_to_check)
```

PURE PYTHON

```
def benchmark(benchmark_maximum, prime_func):
    for i in xrange(benchmark_maximum):
        prime_func(i)
```

PURE PYTHON

```
$ python purepython.py 5
5 is prime.

$ python purepython.py 9
9 is not prime.

$ time python purepython.py -b 1000
…
```

PRIME 5 WAYS

- Pure Python (CPython)
- **Pure C**
- Python with C Module
- Python with C Types
- PyPy

PURE C

```
bool isPrimeNaive(const unsigned int number) {
    /*
    return false for even and 1, true for 2
    */

    unsigned int testMax = primeTestMax(number), divisor;

    for(divisor = 3; divisor <= testMax; divisor += 2)
        if (number % divisor == 0)
            return false;

    return true;
}
```

PURE C

Get unsigned int typed ceil(sqrt(number))

```
bool isPrimeNaive(const unsigned int number) {
    /*
    return false for even and 1, true for 2
    */
    unsigned int testMax = primeTestMax(number), divisor;

    for(divisor = 3; divisor <= testMax; divisor += 2)
        if (number % divisor == 0)
            return false;

    return true;
}
```

PURE C

con_math.c

libcon_math.so

main.c

pure c binary

PURE C

```
$ ./purec 5
5 is prime.

$ ./purec 9
9 is not prime.

$ time ./purec -b 1000
…
```

PRIME 5 WAYS

- Pure Python (CPython)
- Pure C
- **Python with C Module**
- Python with C Types
- PyPy

PYTHON C MODULE

- Write your functions in C (usually just wrapper code)
- Define the mapping between Python and C naming
- Create init function to set up module

PYTHON C MODULE

- **Write your functions in C (usually just wrapper code)**
- Define the mapping between Python and C naming
- Create init function to set up module

PYTHON C MODULE

```
#include <Python.h>
#include <con_math.h>

static PyObject
*con_math_py_is_prime_naive(PyObject *self, PyObject *args)
 {
    unsigned int numberToCheck;

    if (!PyArg_ParseTuple(args, "I", &numberToCheck))
        return NULL;

    bool result = isPrimeNaive(numberToCheck);

    return Py_BuildValue("i", result);
}
```

PYTHON C MODULE

```
#include <Python.h>
#include <con_math.h>

static PyObject
*con_math_py_is_prime_naive(PyObject *self, PyObject *args)
{
    unsigned int numberToCheck;

    if (!PyArg_ParseTuple(args, "I", &numberToCheck))
        return NULL;

    bool result = isPrimeNaive(numberToCheck);

    return Py_BuildValue("i", result);
}
```

PYTHON C MODULE

```
#include <Python.h>
#include <con_math.h>

static PyObject
*con_math_py_is_prime_naive(PyObject *self, PyObject *args)
 {
    unsigned int numberToCheck;

    if (!PyArg_ParseTuple(args, "I", &numberToCheck))
        return NULL;

    bool result = isPrimeNaive(numberToCheck);

    return Py_BuildValue("i", result);
}
```

PYTHON C MODULE

```
#include <Python.h>
#include <con_math.h>

static PyObject
*con_math_py_is_prime_naive(PyObject *self, PyObject *args)
{
    unsigned int numberToCheck;

    if (!PyArg_ParseTuple(args, "I", &numberToCheck))
        return NULL;

    bool result = isPrimeNaive(numberToCheck);

    return Py_BuildValue("i", result);
}
```

PYTHON C MODULE

```
#include <Python.h>
#include <con_math.h>

static PyObject
*con_math_py_is_prime_naive(PyObject *self, PyObject *args)
{
    unsigned int numberToCheck;

    if (!PyArg_ParseTuple(args, "I", &numberToCheck))
        return NULL;

    bool result = isPrimeNaive(numberToCheck);

    return Py_BuildValue("i", result);
}
```

PYTHON C MODULE

```
#include <Python.h>
#include <con_math.h>

static PyObject
*con_math_py_is_prime_naive(PyObject *self, PyObject *args)
{
    unsigned int numberToCheck;

    if (!PyArg_ParseTuple(args, "I", &numberToCheck))
        return NULL;

    bool result = isPrimeNaive(numberToCheck);

    return Py_BuildValue("i", result);
}
```

From con_math.c

PYTHON C MODULE

```
#include <Python.h>
#include <con_math.h>

static PyObject
*con_math_py_is_prime_naive(PyObject *self, PyObject *args)
{
    unsigned int numberToCheck;

    if (!PyArg_ParseTuple(args, "I", &numberToCheck))
        return NULL;

    bool result = isPrimeNaive(numberToCheck);

    return Py_BuildValue("i", result);
}
```

PYTHON C MODULE

- Write your functions in C (usually just wrapper code)
- **Define the mapping between Python and C naming**
- Create init function to set up module

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

- Write your functions in C (usually just wrapper code)
- Define the mapping between Python and C naming
- **Create init function to set up module**

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

```
static PyMethodDef ConMathMethods[] = {
    {"is_prime_naive",  con_math_py_is_prime_naive,
METH_VARARGS,
     "Test if a number is prime."},
    {NULL, NULL, 0, NULL}
};


PyMODINIT_FUNC initcon_math(void) {
    (void) Py_InitModule("con_math", ConMathMethods);
}
```

PYTHON C MODULE

setup.py

```
from distutils.core import setup, Extension

setup(
    ext_modules=[Extension("con_math", ["con_math_py.c",
        "../lib/con_math.c"], include_dirs=['../lib'])],
)
```

PYTHON C MODULE

```
from con_math import is_prime_naive
from pythonlib.wrappers import main


if __name__ == '__main__':
    main(is_prime_naive)
```

C MODULE CF. PURE PYTHON

```
from pythonlib.con_math import is_prime_naive

                    becomes

from con_math import is_prime_naive
```

PYTHON C MODULE

```
$ python python_c_extension.py 5
5 is prime.

$ python python_c_extension.py 9
9 is not prime.

$ time python python_c_extension.py -b 1000
…
```

PRIME 5 WAYS

- Pure Python (CPython)
- Pure C
- Python with C Module
- **Python with C Types**
- PyPy

PYTHON CTYPES

```
from ctypes import CDLL
from pythonlib.wrappers import main


if __name__ == '__main__':
    lib = CDLL('../lib/libcon_math.so')
    main(lib.isPrimeNaive)
```

PYTHON CTYPES

```
from ctypes import CDLL
from pythonlib.wrappers import main


if __name__ == '__main__':
    lib = CDLL('../lib/libcon_math.so')
    main(lib.isPrimeNaive)
```

PYTHON CTYPES

Reference to libcon_math.so

```
lib = CDLL('../lib/libcon_math.so')
```

PYTHON CTYPES

```
from ctypes import CDLL
from pythonlib.wrappers import main

if __name__ == '__main__':
    lib = CDLL('../lib/libcon_math.so')
    main(lib.isPrimeNaive)
```

From libcon_math.so

PYTHON CTYPES

```
$ python python_ctypes.py 5
5 is prime.

$ python python_ctypes.py 9
9 is not prime.

$ time python python_ctypes.py -b 1000
…
```

PRIME 5 WAYS

- Pure Python (CPython)
- Pure C
- Python with C Module
- Python with C Types
- **PyPy**

PYPY

- Download and uncompress
  - (Binaries available for a bunch of different OSs)
- Use anywhere you would have called your **`python`** binary previously
- No changes to your Python code
- Your C libraries may not be supported

PURE PYTHON WITH PYPY

```
$ ~/pypy/bin/pypy purepython.py 5
5 is prime.
```

```
$ ~/pypy/bin/pypy purepython.py 9
9 is not prime.

$ time ~/pypy/bin/pypy purepython.py -b 1000
…
```

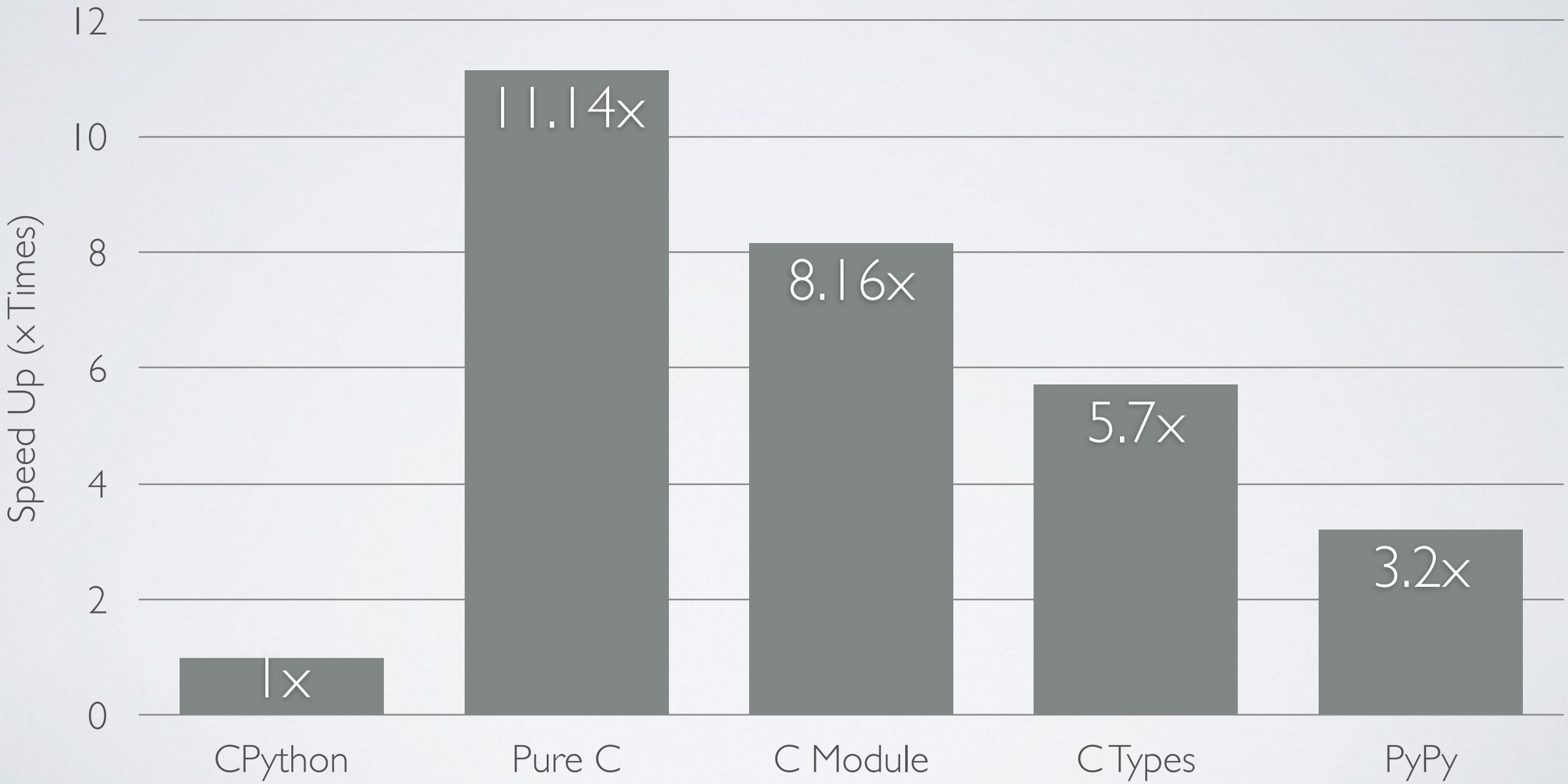
BENCHMARKS

BENCHMARKS

- Use the `-b` flag to benchmark, evaluate each number up to 1,000,000 for primality
- Measure execution time using `time`
  - total time = user + sys
- Run each implementation 10 times
  - `$ for i in {1..10}; do time ./purec -b 1000000; done`
- Take the mean of the results

EXECUTION TIME

SPEED UP


Speed Up (x Times)
12
10
8
6
4
2
0
1x
11.14x
8.16x
5.7x
3.2x
CPython
Pure C
C Module
C Types
PyPy

CONCLUSIONS

PYPY

- Can give you good gains without changes to your Python code
- Provided you don't need to use a library it doesn't support
- Is not a solution for integrating Python with C

CTYPES

- No wrapper C code to write - provided your library already compiles to a shared object
- Simple to use if you're happy without Python exceptions and objects coming back from your library
- Pretty darn fast

C MODULE

- You gotta write a bunch of wrapper C
- Good integration with Python types, exceptions and objects
- Faster than C types, getting closer to Pure C

PURE C

- What are you doing here?

PURE PYTHON

- It's slow
- We love it
- It's the reason we're here

RESOURCES

- Code on GitHub
  https://github.com/beneboy/py-c-integration-example

- Slides and writeup at http://bbit.co.nz/blog/4/