eGenix.com

When performance matters ...

PyCon UK 2014
Coventry, UK

Marc-André Lemburg

(c) 2014 eGenix.com Software, Skills and Services GmbH, info@egenix.com

Speaker Introduction

Marc-André Lemburg

- Python since 1993/1994
- Studied Mathematics
- eGenix.com GmbH
- Python Core Developer
- Python Software Foundation
- EuroPython Society
- Based in Düsseldorf, Germany
- Available for Python Coaching and Consulting

Agenda

- Calls & Loops
- Sorting & Joining
- Lookups & Exceptions
- Conclusion
- Questions

Agenda

- Calls & Loops
- Sorting & Joining
- Lookups & Exceptions
- Conclusion
- Questions

Function calls

f(x)

Function calls

- Python function calls are slow
  - Example:

```
def add(x, y):
    return x+y
```

74 usec

- Python C function calls are faster
  - Example:

```
operator.add
```

56 usec

Method calls

- Python method calls are slow
  - Example:

```
class MyClass:
    def add(self, x, y):
        return x+y
```

  - Timing just the method call

    71 usec

  - Timing method lookup + call

    121 usec

Fastest: Inline operators

- Python operators (x+y) are faster than Python C functions

24 usec

  - To be fair:
    Integer operations are optimized in the Python VM

Calls in Python: Summary

- Python function calls are slow

  74 usec

- Python method calls are slower

  (method lookup + call)

  121 usec

- Python C function calls are faster

  56 usec

- Python operators are even faster

  24 usec

If you have simple types:
Cython generates C speed code

Loops

Loops

- Ranges:

  for i in range(10): ...

  – standard, but there's a catch

- Sequences:

  for i, x in enumerate(seq): ...

  – avoids seq[i] lookups, great for iterators

Loops: Operating on a sequence in Python

- for loop: for i, x in enumerate(seq): l[i] = op(x)

  108 usec

- map function: map(op, seq)

  79 usec

- list comprehension: [op(x) for x in seq]

  75 usec

- generator expression: list (op(x) for x in seq)

  88 usec

op = lambda x: x*2

Loops: Operating on a sequence in C

- for loop: for i, x in enumerate(seq): l[i] = op(x)

  64 usec

- map function: map(op, seq)

  81 usec

- list comprehension: [op(x) for x in seq]

  34 usec

- generator expression: list (op(x) for x in seq)

  46 usec

op = operator.inv

(c) 2014 eGenix.com GmbH, info@egenix.com

Loops over ranges

- Typically done using range()

```
for x in range(1000):
    pass
```

- More dynamic, but less known: xrange()

```
for x in xrange(1000):
    pass
```

Loops over ranges: Static vs. Dynamic

- Looping over range(1000) list

12.400 usec
32,320 bytes

- Looping over xrange(1000) generator

8.940 usec
40 bytes

No wasted memory, less CO_2,
more time with your kids …

(c) 2014 eGenix.com GmbH, info@egenix.com

Loops over ~~ranges~~ xranges

- Obvious choice:

xrange()

… and because it's obvious, it's the default in Python 3:

Python 2	→ Python 3
range()	→ list(range())
xrange()	→ range()

Loops: More helpers

- Collection of loop helpers:

  itertools module

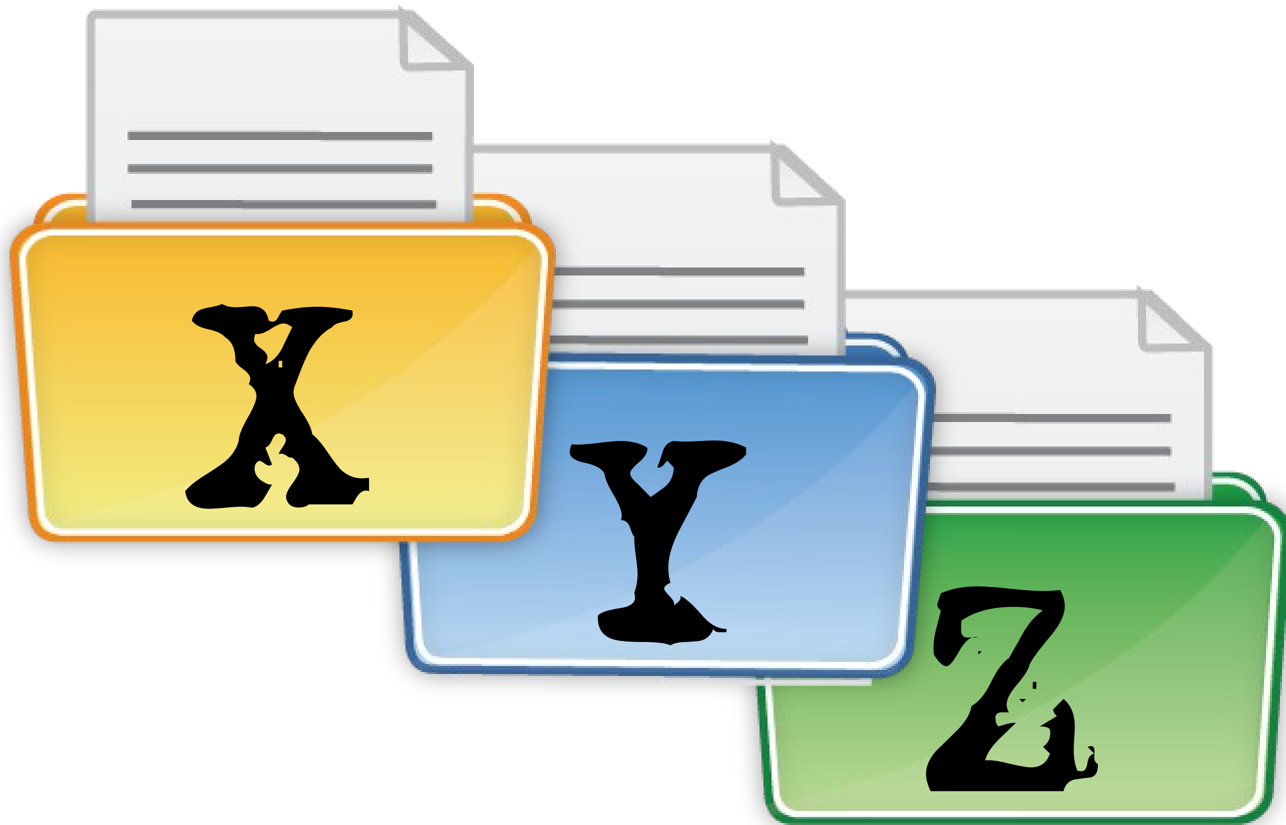
  https://docs.python.org/2.7/library/itertools.html

- If possible, have C code run your loops
  - numpy arrays
  - PIL
  - lxml
  - mxTextTools

Agenda

- Calls & Loops
- Sorting & Joining
- Lookups & Exceptions
- Conclusion
- Questions

Sorting



Sorting: Simple cases

- Lists: l.sort() method

  31 usec

  - returns None
  - changes the list in place

- Any iterable: sorted() builtin

  31 usec

  - returns sorted list copy
  - does not change argument object

Sorting: Decorate-sort-undecorate

- Sort on second item:

```
decorated = [(x[1], x) for x in unsorted_sequence]
decorated.sort()
sorted_sequence = [x for item, x in decorated]
```

103 usec

- Using sorted() and generators:

```
sorted_sequence = [x for (item, x) in sorted(
    (x[1], x) for x in unsorted_sequence)]
```

114 usec

  – Also called the “Schwartzian Transform” in Perl Land

Sorting: Key based

- Sort on second item:
  - l.sort(key=lambda x: x[1])

105 usec

or faster, but less known:

  - l.sort(key=operator.itemgetter(1))

62 usec

  - Use the operator module more often !

Sorting: Key based

- Sort on last_name attribute:
  - l.sort(key=lambda x: x.last_name)

107 usec

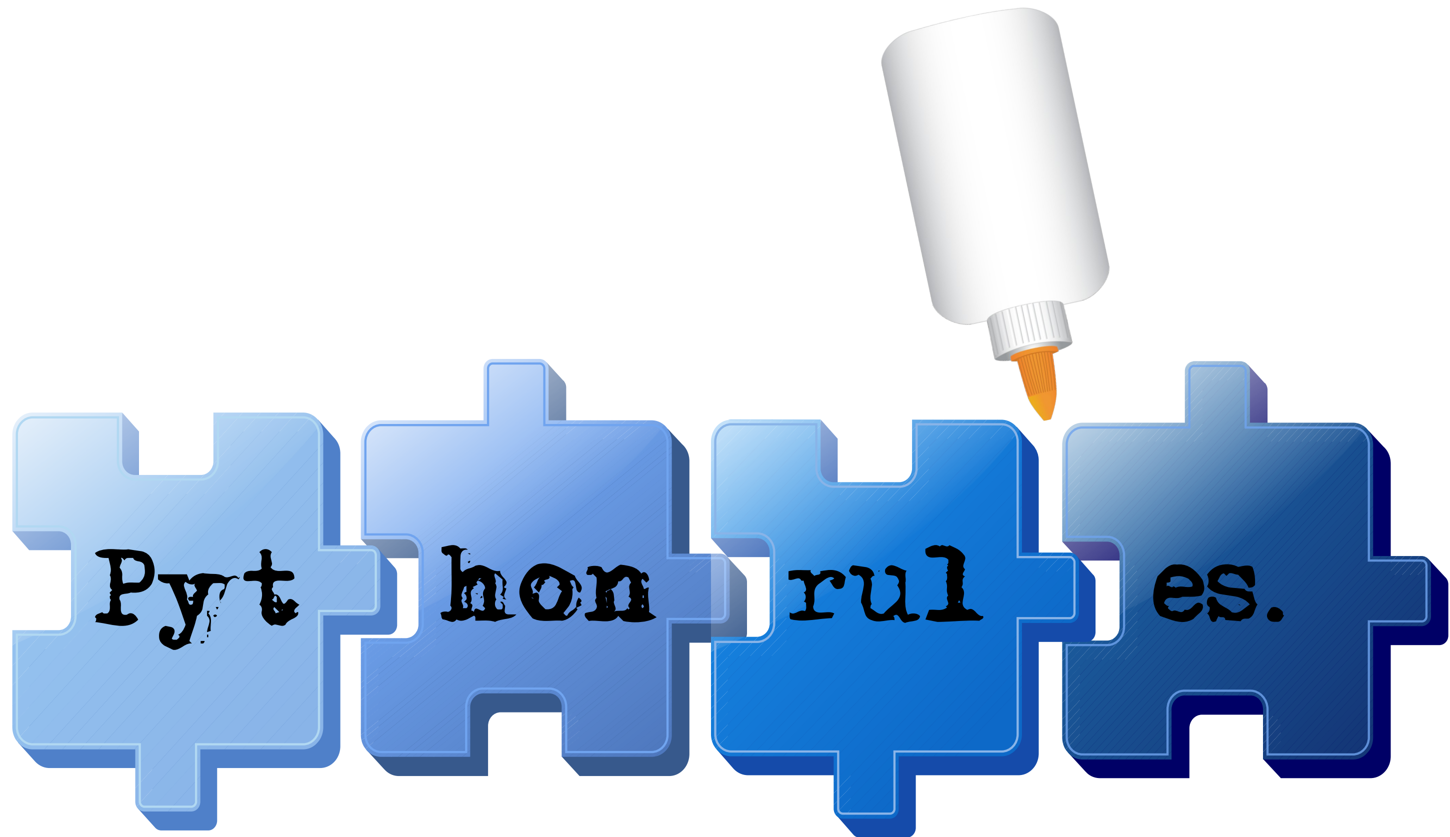
or faster, but less known:

  - l.sort(key=operator.attrgetter('last_name'))

79 usec

  - Use the operator module more often !

Joining strings



Joining strings: Just a few strings

- Using + (concat)
  - Good when concatenating a few strings
  - More readable

18 usec

  - Copies strings together
  - Example: a + b + c + d

- Using ''.join()

16 usec

  - Copies strings, but only once
  - Example: ''.join([a, b, c, d])

Joining strings: Many strings

- Using + (concat)

42 usec

  - Copies strings together
  - Example: a1 + a2 + … + a1000

- Using ''.join()
  - Great for concatenating many strings

9 usec

  - Copies strings, but only once
  - Example: ''.join([a1, a2, …, a1000])

Joining strings: %-formatting

- Using string formatting
  - Few strings

    24 usec
  - Many strings

    20 usec
  - Copies strings, but only once
  - Example: '%s%s%s' % (a, b, c)
  - Looks ugly, not really faster

(c) 2014 eGenix.com GmbH, info@egenix.com

Joining strings: Some other methods

- Using array.fromstring()
  - Slower than + and ''.join()

96 usec

- Using cStringIO.write()
  - Even slower than arrays

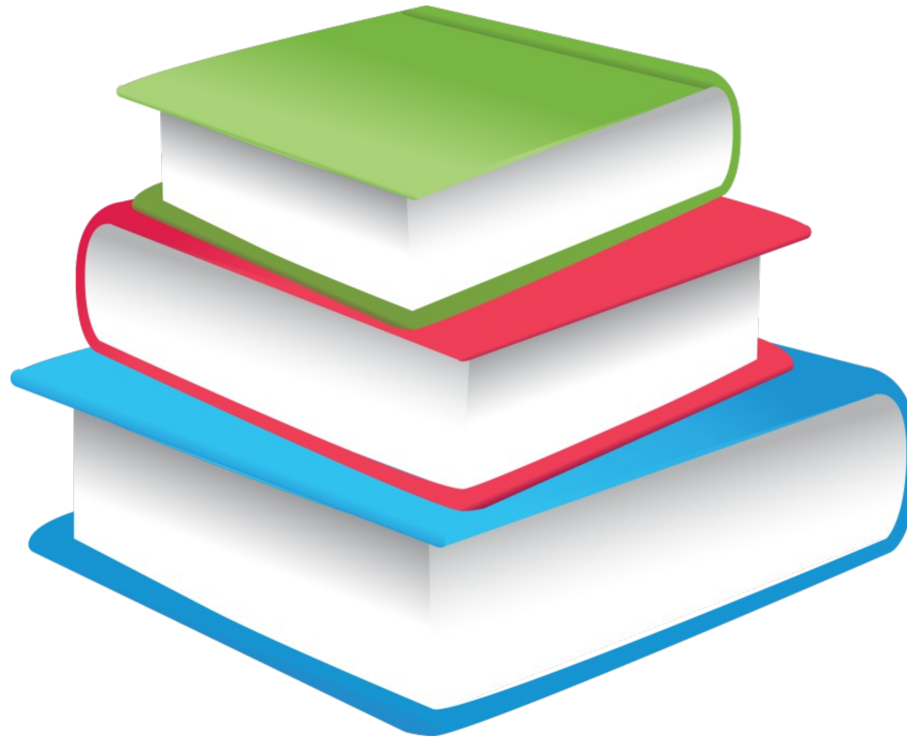
134 usec

  - Not really recommended

Agenda

- Calls & Loops
- Sorting & Joining
- **Lookups & Exceptions**
- Conclusion
- Questions

Dictionary lookups



```
{'question': 'answer'}['question']
```

Dictionary lookups: integer keys faster than strings

- Using integer keys

26 usec

  - Example: d = {1:2, 3:4}
  - Great for storing indexed data

- Using string keys

44 usec

  - Example: d = {'1':2, '3':4}
  - Take longer, because of string comparisons

Dictionary lookups: Speed up string keys

- Using *interned* string keys

27 usec

  - Example: d = {intern('1'):2, intern('3'):4}
  - Fast: Interned string can be compared by pointer
  - Downside: Interned string never get garbage collected
  - In Python 3: sys.intern()

Variable lookups

X = 2
B = “salt”
C = “pepper”

Variable lookups: Locals are faster than globals

- Lookup global variable

16 usec

```
global_a = 'global'
def variable_lookup_global():
    for i in loops:
        x = global_a
```

- Lookup local variable

12 usec

```
def variable_lookup_local():
    local_a = 'local'
    for i in loops:
        x = local_a
```

Global variable lookups: Closer look

- Lookup global variable

85 usec

```
global_a = 'global'
def f(a, b, c):
    x = global_a
    y = global_a
    return y
```

  - Standard Python: performance not really all that bad

Global variable lookups: Keyword argument trick

- Localized global in *keyword argument*

81 usec

```
def f(a, b, c, local_a=global_a):
    x = local_a
    y = local_a
    return y
```

  - Fast: Global gets loaded at function construction time
  - Downside: Can introduce errors

Global variable lookups: Explicit localization in body

- Localized global in function body

90 usec

```
def f(a, b, c):
    local_a = global_a
    x = local_a
    y = local_a
    return y
```

  - Only useful if global is used in tight loops

Variable lookups: Summary

- Local variable lookups faster than globals

  77 usec

- Lookup global variable

  85 usec

- Localized global in keywork argument

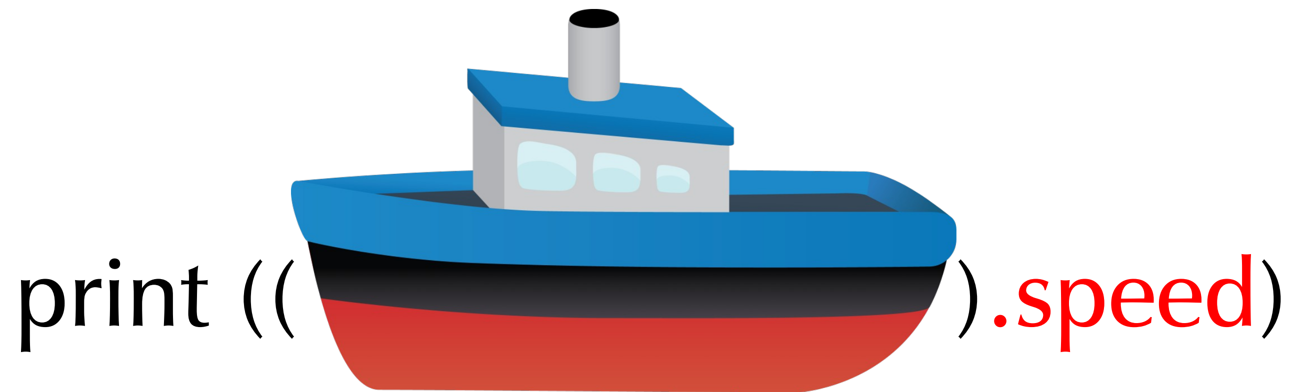
  81 usec

- Localized global in function body

  90 usec

(c) 2014 eGenix.com GmbH, info@egenix.com

Attribute lookups



Attribute lookups

- Class definition:

```
class AttributeObject(object):
    class_attribute = 3
    instance_attribute = None
    def __init__(self):
        self.instance_attribute = 4
```

- Object creation:

```
obj = AttributeObject()
```

Attribute lookups: Results

- Instance attribute lookup

  30 usec

- Class attribute lookup

  30 usec

  - No difference between instance and class attributes
  - Localizing class attributes as instance attributes does not pay off

Attribute lookups: Old-style vs. New-style

- Instance attribute lookup

  old-style classes: 26 usec

  new-style classes: 30 usec

- Class attribute lookup

  old-style classes: 33 usec

  new-style classes: 30 usec

  – There might be some potential for optimization in CPython …

Little known: Slot attributes

- Class definition:

```
class SlotClass(object):
    readonly_slot = 3
    #readwrite_slot is not defined on class
    __slots__ = ('readonly_slot', 'readwrite_slot')
    def __init__(self):
        self.readwrite_slot = 4
```

- Object creation:

```
obj = SlotClass()
```

Slot lookups: Fast and efficient

- Read-only slot lookup

  24 usec

  ```
  x = obj.readonly_slot
  ```

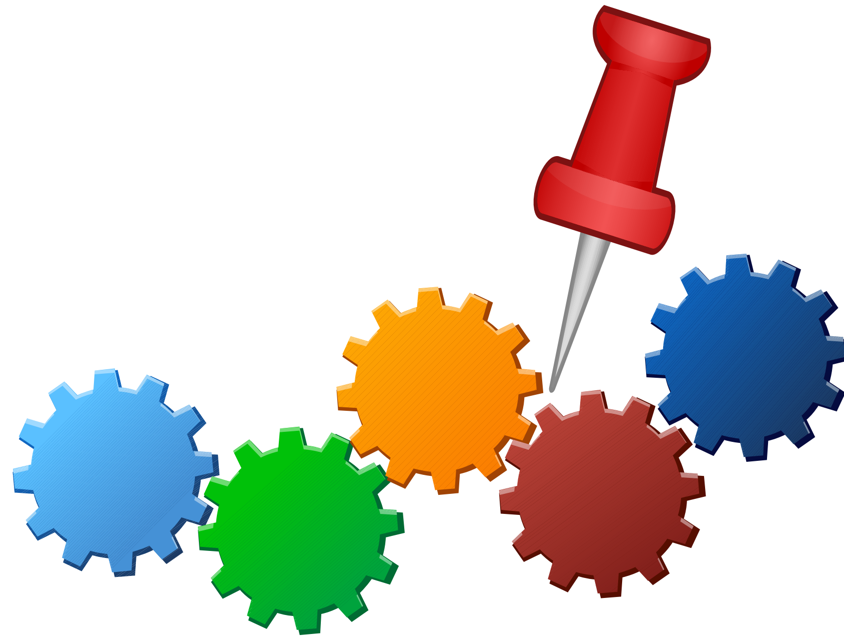
- Read-write slot lookup

  31 usec

  ```
  x = obj.readwrite_slot
  ```

  – Not faster than regular lookups, but:
    Slots save memory – no need for a dictionary !

Exceptions



Exceptions

- Expensive in Python (but cheap in Python's C API)
- Exceptions should only be used for exceptional cases ...

```
# This is not a good idea:
try:
  fails_often()
except ValueError:
  runs_often()
```

Exceptions: Catching exceptions in Python

- Typical use case: attribute introspection

```
try:
    x = obj.attribute
except AttributeError:
    pass
```

- Exception triggers

  621 usec

- Exception does not trigger

  43 usec

Exceptions: Catching exceptions in C

- More efficient method:

  x = getattr(obj, 'attribute', None)

- Exception triggers

  269 usec

- Exception does not trigger

  74 usec

Exceptions: Summary

- Exception triggers

try-except: 621 usec
getattr(): 269 usec

- Exception does not trigger

try-except: 43 usec
getattr(): 74 usec

  - For exceptions that trigger often, try to process them at C level

Agenda

- Calls & Loops
- Sorting & Joining
- Lookups & Exceptions
- Conclusion
- Questions

Conclusion

Higher-level performance tips

- Use efficient data structures
- Use efficient algorithms

  → lots of research available – free to use :-)

- Refactor/reorganize/profile your code regularly
- Move your loops to Python C extensions
- Rewrite hotspots in Cython or C

Resources

- Performance tests/tools are available on Github:

  https://github.com/egenix/when-performance-matters

- Contact me if you need help (mal@egenix.com)

Agenda

- Calls & Loops
- Sorting & Joining
- Lookups & Exceptions
- Conclusion
- Questions

Questions

Thank you for listening

Beautiful is better than ugly.

Contact

eGenix.com Software, Skills and Services GmbH
Marc-André Lemburg
Pastor-Löh-Str. 48
D-40764 Langenfeld
Germany

eMail: mal@egenix.com
Phone: +49 211 9304112
Fax: +49 211 3005250
Web: http://www.egenix.com/