Writing Faster Python 3

Sebastian Witowski

switowski.com | @SebaWitowski

Writing Faster Python 3

Sebastian Witowski - Writing faster Python

93K views • 5 years ago

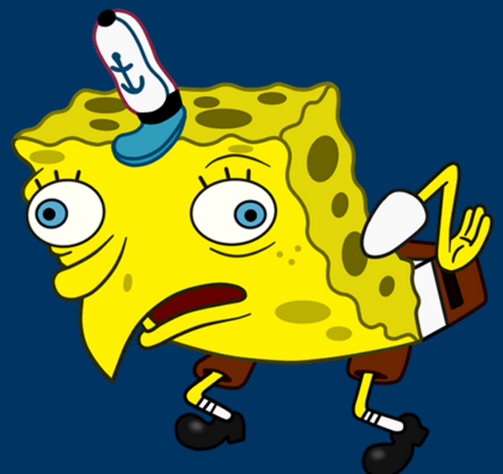
EuroPython Conference

Presentation on how you can **write faster Python** in your daily ...

SETUP | #2 FILTER A LIST | ... 8 moments

Why are you using Python?!

It's so slow!



Python is slow

Python is ~~slow~~ slower

Python was not optimised for the runtime speed.

It was optimised for development speed.

Why is Python slower?

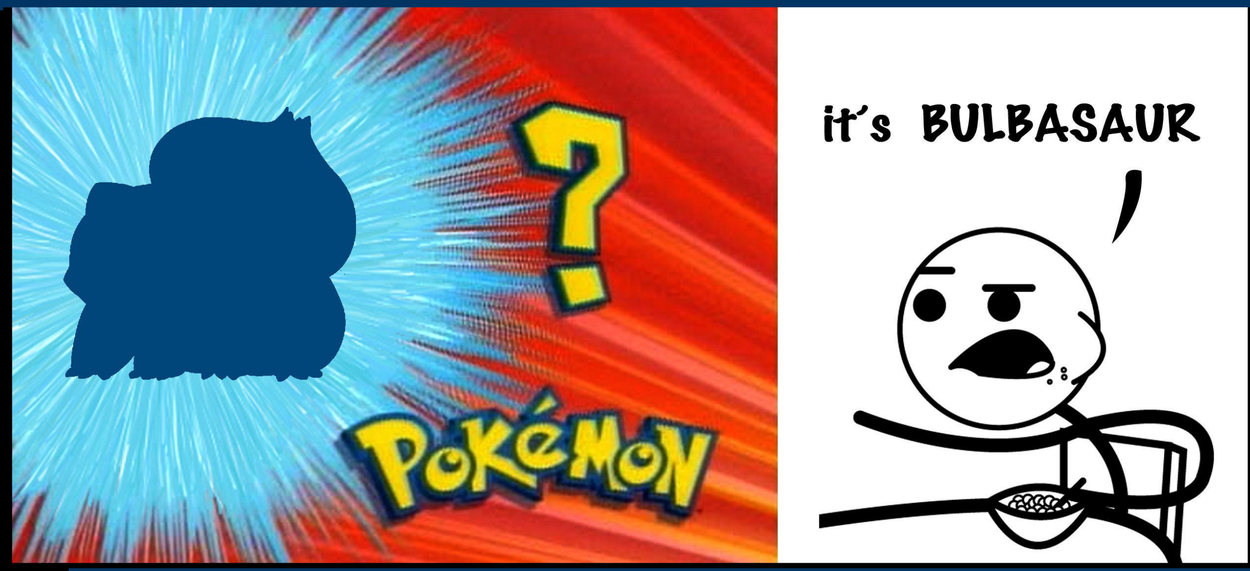
Python is dynamic

Python is dynamic

```
a = "hello"
...
a = 42
...
a = [1,2,3]
...
a = pd.DataFrame()
...
```

Python is dynamic

```
a = "hello"
...
a = 42
...
a = [1,2,3]
...
a = pd.DataFrame()
...
```



Python is dynamic

```
a = "hello"
...
a = 42
...
a = [1,2,3]
...
a = pd.DataFrame()
...
```

Why is Python slow?

Anthony Shaw

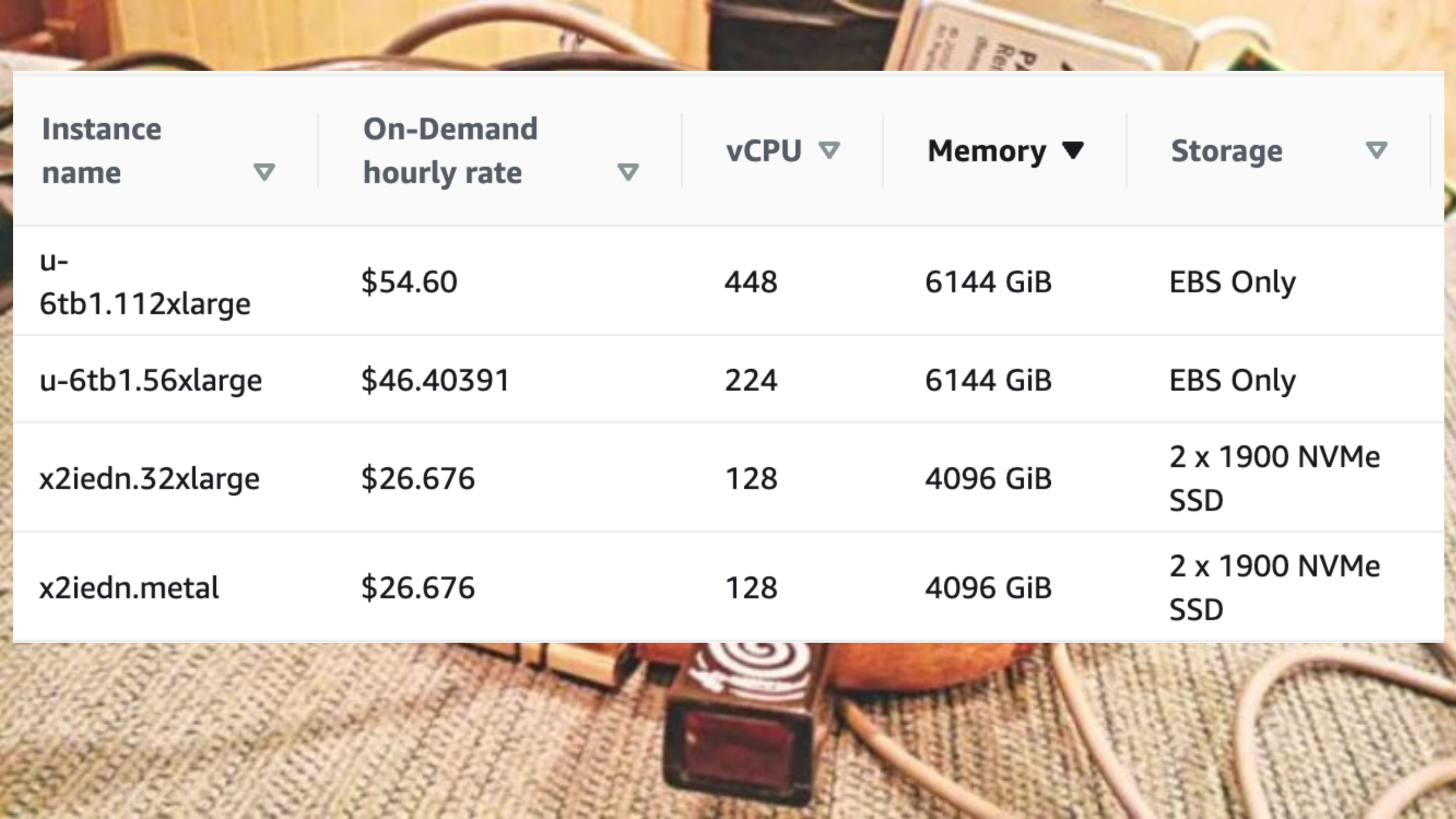
https://www.youtube.com/watch?v=I4nkgdVZFA

How to speed up Python code?

How to speed up Python code?

- Get faster hardware


AVAYA
PARTNER
Remote Access PC Card
GIGABYTE



Instance name	On-Demand hourly rate	vCPU	Memory ▼	Storage
u-6tb1.112xlarge	$54.60	448	6144 GiB	EBS Only
u-6tb1.56xlarge	$46.40391	224	6144 GiB	EBS Only
x2iedn.32xlarge	$26.676	128	4096 GiB	2 x 1900 NVMe SSD
x2iedn.metal	$26.676	128	4096 GiB	2 x 1900 NVMe SSD

Instance name	On-Demand hourly rate	vCPU	Memory ▼	Storage
u-6tb1.112xlarge	$54.60	448	6144 GiB	EBS Only
u-6tb1.56xlarge	$46.40391	224	6144 GiB	EBS Only
x2iedn.32xlarge	$26.676	128	4096 GiB	2 x 1900 NVMe SSD
x2iedn.metal	$26.676	128	4096 GiB	2 x 1900 NVMe SSD
r6gd.4xlarge	$0.9216	16	128 GiB	1 x 950 NVMe SSD

How to speed up Python code?

- Get faster hardware
- Use a different interpreter

How to speed up Python code?

- Get faster hardware
- Use a different interpreter

Pyston

Pyjion

Cinder

GraalPython

How to speed up Python code?

- Get faster hardware
- Use a different interpreter
- Numpy / numba

How to speed up Python code?

- Get faster hardware
- Use a different interpreter
- Numpy / numba
- Update your Python version

Python 3.10

Optimizations

- Constructors `str()`, `bytes()` and `bytearray()` are now faster (around 30–40% for small objects). (Contributed by Serhiy Storchaka in bpo-41334.)
- The `runpy` module now imports fewer modules. The `python3 -m module-name` command startup time is 1.4x faster in average. On Linux, `python3 -I -m module-name` imports 69 modules on Python 3.9, whereas it only imports 51 modules (-18) on Python 3.10. (Contributed by Victor Stinner in bpo-41006 and bpo-41718.)
- The `LOAD_ATTR` instruction now uses new "per opcode cache" mechanism. It is about 36% faster now for regular attributes and 44% faster for slots. (Contributed by Pablo Galindo and Yury Selivanov in bpo-42093 and Guido van Rossum in bpo-42927, based on ideas implemented originally in PyPy and MicroPython.)
- When building Python with `--enable-optimizations` now `-fno-semantic-interposition` is added to both the compile and link line. This speeds builds of the Python interpreter created with `--enable-shared` with `gcc` by up to 30%. See this article for more details. (Contributed by Victor Stinner and Pablo Galindo in bpo-38980.)
- Use a new output buffer management code for `bz2` / `lzma` / `zlib` modules, and add `.readall()` function to `_compression.DecompressReader` class. bz2 decompression is now 1.09x ~ 1.17x faster, lzma decompression 1.20x ~ 1.32x faster, `GzipFile.read(-1)` 1.11x ~ 1.18x faster. (Contributed by Ma Lin, reviewed by Gregory P. Smith, in bpo-41486)
- When using stringized annotations, annotations dicts for functions are no longer created when the function is created. Instead, they are stored as a tuple of strings, and the function object lazily converts this into the annotations dict on demand. This optimization cuts the CPU time needed to define an annotated function by half. (Contributed by Yurii Karabas and Inada Naoki in bpo-42202)
- Substring search functions such as `str1 in str2` and `str2.find(str1)` now sometimes use Crochemore & Perrin's "Two-Way" string searching algorithm to avoid quadratic behavior on long strings. (Contributed by Dennis Sweeney in bpo-41972)

How to speed up Python code?

- Get faster hardware
- Use a different interpreter
- Numpy / numba
- Update your Python version
- Better algorithms and data structures

```python
# example.py

total = 0
def compute_sum_of_powers():
    global total
    for x in range(1_000_000):
        total = total + x*x

compute_sum_of_powers()
print(total)
```

```python
# example.py

total = 0
def compute_sum_of_powers():
    global total
    for x in range(1_000_000):
        total = total + x*x

compute_sum_of_powers()
print(total)
```

```
$ ipython
In [1]: %time %run example.py
333332833333500000
CPU times: user 70.8 ms, sys: 2.33 ms, total: 73.1 ms
Wall time: 72.8 ms
```

```python
# example.py

total = 0
def compute_sum_of_powers():
    global total
    for x in range(1_000_000):
        total = total + x*x
```

Not the best way to measure the execution time!

```
In [1]: %time %run example.py
333332833333500000
CPU times: user 70.8 ms, sys: 2.33 ms, total: 73.1 ms
Wall time: 72.8 ms
```

Wait, IPython can do that?!

Sebastian Witowski

https://www.youtube.com/watch?v=3i6db5zX3Rw

```python
# example.py

total = 0
def compute_sum_of_powers():
    global total
    for x in range(1_000_000):
        total = total + x*x

compute_sum_of_powers()
print(total)
```

```
$ ipython
In [1]: %time %run example.py
333332833333500000
CPU times: user 70.8 ms, sys: 2.33 ms, total: 73.1 ms
Wall time: 72.8 ms
```

```python
# example.py

total = 0
def compute_sum_of_powers():
    global total
    for x in range(1_000_000):
        total = total + x*x

compute_sum_of_powers()
print(total)
```

```python
# example2.py

def compute_sum_of_powers():
    total = 0
    for x in range(1_000_000):
        total = total + x*x
    return total

total = compute_sum_of_powers()
print(total)
```

63.4 msec (from 72.8)

```python
# example2.py

def compute_sum_of_powers():
    total = 0
    for x in range(1_000_000):
        total = total + x*x
    return total

total = compute_sum_of_powers()
print(total)
```

63.4 msec (from 72.8)

```python
# example3.py

def compute_sum_of_powers():
    return sum([n * n for n in range(1_000_000)])

total = compute_sum_of_powers()
print(total)
```

59.8 msec (from 63.4)

```python
# example3.py

def compute_sum_of_powers():
    return sum([n * n for n in range(1_000_000)])

total = compute_sum_of_powers()
print(total)
```

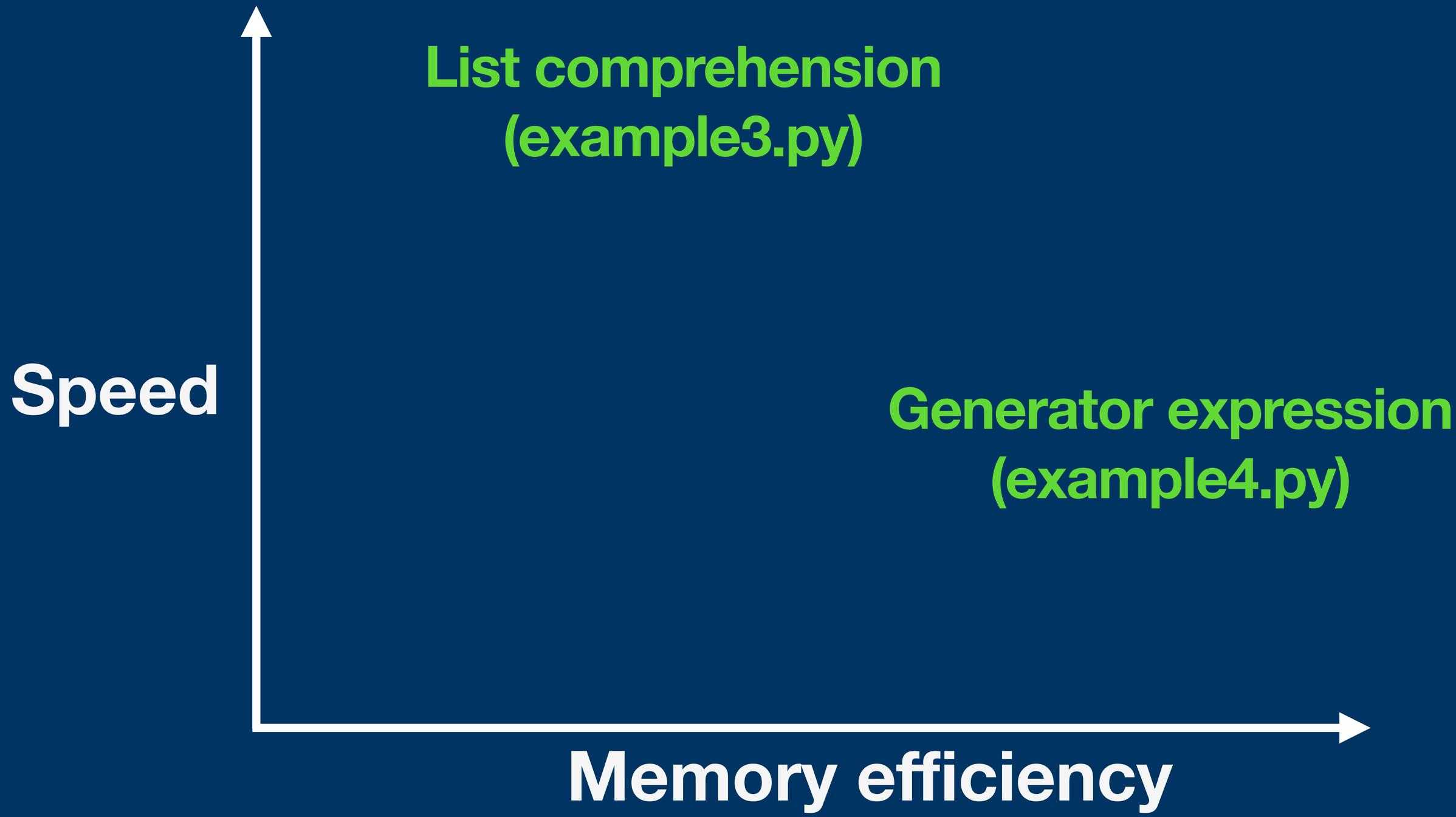
59.8 msec (from 63.4)

```python
# example4.py

def compute_sum_of_powers():
    return sum(n * n for n in range(1_000_000))

total = compute_sum_of_powers()
print(total)
```

62.5 msec (from 59.8)


List comprehension
(example3.py)
Speed
Generator expression
(example4.py)
Memory efficiency

```
$ pip install memory_profiler  # install memory profiler...
$ ipython
In [1]: %load_ext memory_profiler  # ...and activate it

In [2]: %memit sum([n * n for n in range(1_000_000)])
peak memory: 119.39 MiB, increment: 49.20 MiB

In [3]: %memit sum(n * n for n in range(1_000_000))
peak memory: 84.75 MiB, increment: 0.00 MiB
```

```python
# example2_numba.py
from numba import jit  # pip install numba

@jit
def compute_sum_of_powers():
    total = 0
    for x in range(1_000_000):
        total = total + x*x
    return total

total = compute_sum_of_powers()
print(total)
```

34.4 msec (from 63.4 for example2.py)

```python
# example3.py

def compute_sum_of_powers():
    return sum([n * n for n in range(1_000_000)])

total = compute_sum_of_powers()
print(total)
```

```python
# example5.py
import numpy

def compute_sum_of_powers():
    return sum([n * n for n in range(1_000_000)])

total = compute_sum_of_powers()
print(total)
```

```python
# example5.py
import numpy

def compute_sum_of_powers():
    numbers = numpy.arange(1_000_000)
    powers = numpy.power(numbers, 2)
    return numpy.sum(powers)

total = compute_sum_of_powers()
print(total)
```

```python
# example5.py
import numpy

def compute_sum_of_powers():
    numbers = numpy.arange(1_000_000)
    powers = numpy.power(numbers, 2)
    return numpy.sum(powers)

total = compute_sum_of_powers()
print(total)
```

57 msec (from 59.8)

```
$ ipython
In [1]: %time %run example5.py
333332833333500000
CPU times: user 50.7 ms, sys: 8.18 ms, total: 58.9 ms
Wall time: 57 ms # from 59.8 ms

In [2]: %time %run example5.py
333332833333500000
CPU times: user 5.77 ms, sys: 5.84 ms, total: 11.6 ms
Wall time: 9.87 ms
```

```python
# example5.py
import numpy

def compute_sum_of_powers():
    numbers = numpy.arange(1_000_000)
    powers = numpy.power(numbers, 2)
    return numpy.sum(powers)

total = compute_sum_of_powers()
print(total)
```

9.87 msec (from 59.8)

`example.py` improvements

- Local variable

`example.py` improvements

- Local variable
- Built-in function (itertools, collections)

`example.py` improvements

- Local variable
- Built-in function (itertools, collections)
- List comprehension instead of a loop
  - Generator expression for lower memory usage

`example.py` improvements

- Local variable
- Built-in function (itertools, collections)
- List comprehension instead of a loop
  - Generator expression for lower memory usage
- numpy - dedicated library for scientific computing

`example.py` improvements

- Local variable
- Built-in function (itertools, collections)
- List comprehension instead of a loop
  - Generator expression for lower memory usage
- numpy - dedicated library for scientific computing
- numba - JIT decorator for easy wins

Source code optimization

Code repository

github.com/switowski/writing-faster-python3

Benchmarks setup

- Python 3.10.4
- `PYTHONDONTWRITEBYTECODE` set to 1
- `python -m timeit -s "from my_module import function" "function()"`
- Machine: 14-inch Macbook Pro (2021) with 16GB of RAM, M1 with 10 CPU cores and 16 GPU cores

Benchmarks setup

Your numbers will be different.

But "faster" code examples will still run faster than "slower" ones.

1. Permission vs. forgiveness

```
import os

if os.path.exists("myfile.txt"):
    with open("myfile.txt") as input_file:
        return input_file.read()
```

1. Permission vs. forgiveness

```
import os

if os.path.exists("myfile.txt"):
    if os.access("path/to/file.txt", os.R_OK):
        with open("myfile.txt") as input_file:
            return input_file.read()
```

1. Permission vs. forgiveness

```
import os

if os.path.exists("myfile.txt"):
    if os.access("path/to/file.txt", os.R_OK):
        with open("myfile.txt") as input_file:
            return input_file.read()
```

vs.

```
try:
    with open("path/to/file.txt", "r") as input_file:
        return input_file.read()
except IOError:
    # Handle the error or just ignore it
    pass
```

1. Permission vs. forgiveness

```
# permission_vs_forgiveness.py

class BaseClass:
    hello = "world"

class Foo(BaseClass):
    pass

FOO = Foo()

FOO.hello
```

```python
# permission_vs_forgiveness.py

class BaseClass:
    hello = "world"

class Foo(BaseClass):
    pass

FOO = Foo()

# Ask for permission
def test_permission():
    if hasattr(FOO, "hello"):
        FOO.hello

# Ask for forgiveness
def test_forgiveness():
    try:
        FOO.hello
    except AttributeError:
        pass
```

```
$ python -m timeit -s "from permission_vs_forgiveness
import test_permission" "test_permission()"
5000000 loops, best of 5: 71.1 nsec per loop

$ python -m timeit -s "from permission_vs_forgiveness
import test_forgiveness" "test_forgiveness()"
5000000 loops, best of 5: 61.6 nsec per loop
```

71.1 / 61.6 = 1.15

Asking for permission is ~15% slower.

1.1 Permission vs. forgiveness
More than 1 attribute

```python
# permission_vs_forgiveness2.py

class BaseClass:
    hello = "world"
    bar = "world"
    baz = "world"

class Foo(BaseClass):
    pass

FOO = Foo()

# Ask for permission
def test_permission():
    if hasattr(FOO, "hello") and hasattr(FOO, "bar") and hasattr(FOO, "baz"):
        FOO.hello
        FOO.bar
        FOO.baz

# Ask for forgiveness
def test_forgiveness():
    try:
        FOO.hello
        FOO.bar
        FOO.baz
    except AttributeError:
        pass
```

```
$ python -m timeit -s "from permission_vs_forgiveness2
import test_permission" "test_permission()"
2000000 loops, best of 5: 151 nsec per loop

$ python -m timeit -s "from permission_vs_forgiveness2
import test_forgiveness" "test_forgiveness()"
5000000 loops, best of 5: 82.9 nsec per loop
```

151/82.9 = 1.82

Asking for permission with 3 attributes is ~82% slower.

Is asking for forgiveness always the best choice?

1.3 Permission vs. forgiveness
Missing attribute

```python
# permission_vs_forgiveness3.py

class BaseClass:
    hello = "world"
    # bar = "world"
    baz = "world"

class Foo(BaseClass):
    pass

FOO = Foo()

# Ask for permission
def test_permission():
    if hasattr(FOO, "hello") and hasattr(FOO, "bar") and hasattr(FOO, "baz"):
        FOO.hello
        FOO.bar
        FOO.baz

# Ask for forgiveness
def test_forgiveness():
    try:
        FOO.hello
        FOO.bar
        FOO.baz
    except AttributeError:
        pass
```

```
$ python -m timeit -s "from permission_vs_forgiveness3
import test_permission" "test_permission()"
5000000 loops, best of 5: 81.4 nsec per loop

$ python -m timeit -s "from permission_vs_forgiveness3
import test_forgiveness" "test_forgiveness()"
1000000 loops, best of 5: 309 nsec per loop
```

309/81.4 = 3.8

Asking for forgiveness with a missing attributes is almost 4 times as slow as asking for permission!


Well, well, well,
how the turntables...

1. Permission vs. forgiveness

"*Is it more likely that my code will throw an exception?*"

2. Find element in a collection

```python
# find_element.py

def while_loop():
    number = 1
    while True:
        # You don't need to use parentheses, but they improve readability
        if (number % 42 == 0) and (number % 43 == 0):
            return number  # That's 1806
        number += 1
```

2. Find element in a collection

```python
# find_element.py

def while_loop():
    number = 1
    while True:
        # You don't need to use parentheses, but they improve readability
        if (number % 42 == 0) and (number % 43 == 0):
            return number  # That's 1806
        number += 1
```

```python
from itertools import count

def for_loop():
    for number in count(1):
        if (number % 42 == 0) and (number % 43 == 0):
            return number
```

2. Find element in a collection

```
# find_element.py

def while_loop():
    number = 1
    while True:
        # You don't need to use parentheses, but they improve readability
        if (number % 42 == 0) and (number % 43 == 0):
            return number  # That's 1806
        number += 1
```

59.4 usec (59.4/47 = 1.26)

```
from itertools import count

def for_loop():
    for number in count(1):
        if (number % 42 == 0) and (number % 43 == 0):
            return number
```

47 usec

2. Find element in a collection

```
from itertools import count

def for_loop_count():
    for number in count(1):
        if (number % 42 == 0) and (number % 43 == 0):
            return number
```

47 usec

```
def list_comprehension():
    return [n for n in range(1, 10_000) if (n % 42 == 0) and (n % 43 == 0)][0]
```

2. Find element in a collection

```
from itertools import count

def for_loop_count():
    for number in count(1):
        if (number % 42 == 0) and (number % 43 == 0):
            return number
```

47 usec

```
def list_comprehension():
    return [n for n in range(1, 10_000) if (n % 42 == 0) and (n % 43 == 0)][0]
```

254 usec (254/47 = 5.4)

2. Find element in a collection

```
from itertools import count

def for_loop_count():
    for number in count(1):
        if (number % 42 == 0) and (number % 43 == 0):
            return number
```

47 usec

```
def generator():
    return next(n for n in count(1) if (n % 42 == 0) and (n % 43 == 0))
```

2. Find element in a collection

```
from itertools import count

def for_loop_count():
    for number in count(1):
        if (number % 42 == 0) and (number % 43 == 0):
            return number
```

47 usec

```
def generator():
    return next(n for n in count(1) if (n % 42 == 0) and (n % 43 == 0))
```

45.7 usec (47/45.7 = 1.03)

2. Find element in a collection

Generator expression - fast, concise, and memory-efficient.

For loop - for complex "if" statements.

3. Filter a list

```
# filter_list.py
NUMBERS = range(1_000_000)

def test_loop():
    odd = []
    for number in NUMBERS:
        if number % 2:
            odd.append(number)
    return odd
```

3. Filter a list

```
# filter_list.py
NUMBERS = range(1_000_000)

def test_loop():
    odd = []
    for number in NUMBERS:
        if number % 2:
            odd.append(number)
    return odd
```

33.5 msec

3. Filter a list

```
# filter_list.py
NUMBERS = range(1_000_000)

def test_loop():
    odd = []
    for number in NUMBERS:
        if number % 2:
            odd.append(number)
    return odd
```

33.5 msec

```
def test_filter():
    return list(filter(lambda x: x % 2, NUMBERS))
```

3. Filter a list

```
# filter_list.py
NUMBERS = range(1_000_000)

def test_loop():
    odd = []
    for number in NUMBERS:
        if number % 2:
            odd.append(number)
    return odd
```

33.5 msec

```
def test_filter():
    return list(filter(lambda x: x % 2, NUMBERS))
```

49.9 msec (49.9/33.5 = 1.49)

3. Filter a list

```
# filter_list.py
NUMBERS = range(1_000_000)

def test_loop():
    odd = []
    for number in NUMBERS:
        if number % 2:
            odd.append(number)
    return odd
```

33.5 msec

```
def test_filter():
    return list(filter(lambda x: x % 2, NUMBERS))
```

49.9 msec (49.9/33.5 = 1.49)

```
def test_comprehension():
    return [number for number in NUMBERS if number % 2]
```

3. Filter a list

```
# filter_list.py
NUMBERS = range(1_000_000)

def test_loop():
    odd = []
    for number in NUMBERS:
        if number % 2:
            odd.append(number)
    return odd
```

33.5 msec (33.5/25.9 = 1.29)

```
def test_filter():
    return list(filter(lambda x: x % 2, NUMBERS))
```

49.9 msec (49.9/25.9 = 1.92)

```
def test_comprehension():
    return [number for number in NUMBERS if number % 2]
```

25.9 msec

3. Filter a list

List comprehension - when you need a list.

Filter - when you need an iterator.

For loop - for complex conditions.

3. Filter a list

```
# filter_list.py
NUMBERS = range(1_000_000)

def test_loop():
    odd = []
    for number in NUMBERS:
        if number % 2:
            odd.append(number)
    return odd
```

33.5 msec (33.5/25.9 = 1.29)

```
def test_filter():
    return list(filter(lambda x: x % 2, NUMBERS))
```

49.9 msec (49.9/25.9 = 1.92)

```
def test_comprehension():
    return [number for number in NUMBERS if number % 2]
```

25.9 msec

3. Filter a list

List comprehension - when you need a list.

Filter - when you need an iterator.

For loop - for complex conditions.

4. Membership testing

```
# membership.py

MILLION_NUMBERS = list(range(1_000_000))

def test_for_loop(number):
    for item in MILLION_NUMBERS:
        if item == number:
            return True
    return False
```

4. Membership testing

```
# membership.py

MILLION_NUMBERS = list(range(1_000_000))

def test_for_loop(number):
    for item in MILLION_NUMBERS:
        if item == number:
            return True
    return False
```

```
def test_in(number):
    return number in MILLION_NUMBERS
```

4. Membership testing

```
# membership.py

MILLION_NUMBERS = list(range(1_000_000))

def test_for_loop(number):
    for item in MILLION_NUMBERS:
        if item == number:
            return True
    return False
```

```
def test_in(number):
    return number in MILLION_NUMBERS
```

test_for_loop(42) vs. test_in(42)

591 nsec vs. **300** nsec (591/300 = 1.97)

test_for_loop(999_958) vs. test_in(999_958)

12.7 msec vs. **6.02** msec (12.7/6.02 = 2.11)

test_for_loop(-5) vs. test_in(-5)

12.7 msec vs. **5.87** msec (591/300 = 2.16)

4. Membership testing

```
# membership2.py

MILLION_NUMBERS = list(range(1_000_000))

def test_in(number):
    return number in MILLION_NUMBERS
```

```
MILLION_NUMBERS_SET = set(MILLION_NUMBERS)

def test_in_set(number):
    return number in MILLION_NUMBERS_SET
```

4. Membership testing

```
# membership2.py

MILLION_NUMBERS = list(range(1_000_000))

def test_in(number):
    return number in MILLION_NUMBERS
```

```
MILLION_NUMBERS_SET = set(MILLION_NUMBERS)

def test_in_set(number):
    return number in MILLION_NUMBERS_SET
```

test_in(42) vs. test_in_set(42)

301 nsec vs. **45.9** nsec (301/45.9 = 6.56)

test_in(999_958) vs. test_in_set(999_958)

6.04 msec vs. **51.5** nsec

(6040000/51.5 = **117,282**)

test_in(-5) vs. test_in_set(-5)

5.87 msec vs. **46.1** nsec

(5870000/46.1 = 127,332)

4. Membership testing

```
# membership2.py

MILLION_NUMBERS = list(range(1_000_000))

def test_in(number):
    return number in MILLION_NUMBERS

MILLION_NUMBERS_SET = set(MILLION_NUMBERS)

def test_in_set(number):
    return number in MILLION_NUMBERS_SET
```

Ok, but let's try without cheating this time

test_in(42) vs. test_in_set(42)

301 nsec vs. 45.9 nsec (301/45.9 = 6.56)

test_in(999_958) vs. test_in_set(999_958)

6.64 msec vs. 51.5 nsec

(6040000/51.5 = **117,282**)

test_in(-5) vs. test_in_set(-5)

5.87 msec vs. 46.1 nsec

(5870000/46.1 = 127,332)

4. Membership testing

```
# membership2.py

MILLION_NUMBERS = list(range(1_000_000))

def test_in(number):
    return number in MILLION_NUMBERS
```

```
MILLION_NUMBERS_SET = set(MILLION_NUMBERS)

def test_in_set(number):
    return number in MILLION_NUMBERS_SET
```

```
def test_in_set_proper(number):
    return number in set(MILLION_NUMBERS)
```

4. Membership testing

```
# membership2.py

MILLION_NUMBERS = list(range(1_000_000))

def test_in(number):
    return number in MILLION_NUMBERS
```

```
MILLION_NUMBERS_SET = set(MILLION_NUMBERS)

def test_in_set(number):
    return number in MILLION_NUMBERS_SET
```

```
def test_in_set_proper(number):
    return number in set(MILLION_NUMBERS)
```

test_in(42) vs. test_in_set_proper(42)

301 nsec vs. **11.8** msec

(11800000/301 = 39,203)

test_in(999_958) vs. test_in_set_proper(999_958)

6.04 msec vs. **11.9** msec

(11.9/6.04 = 1.97)

test_in(-5) vs. test_in_set_proper(-5)

5.87 msec vs. **11.8** msec

(11.8/5.87 = 2.01)

4. Membership testing

```
# membership2.py

MILLION_NUMBERS = list(range(1_000_000))

def test_in(number):
    return number in MILLION_NUMBERS
```

```
MILLION_NUMBERS_SET = set(MILLION_NUMBERS)

def test_in_set(number):
    return number in MILLION_NUMBERS_SET
```

```
def test_in_set_proper(number):
    return number in set(MILLION_NUMBERS)
```

test_in(42) vs. test_in_set_proper(42)

301 nsec vs. **11.8** msec

(11800000/301 = 39,203)

test_in(999_958) vs. test_in_set_proper(999_958)

6.04 msec vs. **11.9** msec

(11.9/6.04 = 1.97)

test_in(-5) vs. test_in_set_proper(-5)

5.87 msec vs. **11.8** msec

(11.8/5.87 = 2.01)

4. Membership testing

For loop - bad

"in" operator - good

Average lookup time: O(n) for list O(1) for set

Converting list to a set is slow

*Set is not a drop-in replacement for a list!

https://wiki.python.org/moin/TimeComplexity

5. dict() vs {}

5. dict() vs {}

```
$ python -m timeit "a = dict()"
```

38.3 nsec (38.3/14 = 2.7)

```
$ python -m timeit "a = {}"
```

14 nsec

5. dict() vs {}

```
In [1]: from dis import dis

In [2]: dis("dict()")
  1           0 LOAD_NAME                0 (dict)
              2 CALL_FUNCTION            0
              4 RETURN_VALUE

In [3]: dis("{}")
  1           0 BUILD_MAP                0
              2 RETURN_VALUE
```

5. dict() vs {}

```
def dict(*args, **kwargs):
    # Happy debugging ;)
    return list([1, 2, 3])
```

5. dict() vs {}

```
In [1]: from dis import dis

In [2]: dis("dict()")
  1           0 LOAD_NAME                0 (dict)
              2 CALL_FUNCTION            0
              4 RETURN_VALUE

In [3]: dis("{}")
  1           0 BUILD_MAP                0
              2 RETURN_VALUE
```

5. dict() vs {}

Literal syntax: `{}`, `[]`, `()` is faster than calling a function: `dict()`, `list()`, `tuple()`

`dis` module shows you what runs "under the hood"

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

```
def test_list_comprehension():
    unique = []
    [unique.append(n) for n in DUPLICATES if n not in unique]
    return unique
```

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

```
def test_list_comprehension():
    unique = []
    [unique.append(n) for n in DUPLICATES if n not in unique]
    return unique
```

315 ms

6. Remove duplicates

```python
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```python
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

~~def test_list_comprehension():~~
~~   unique = []~~
~~   [unique.append(n) for n in DUPLICATES if n not in unique]~~
~~   return unique~~

Don't use list comprehension **only** for the side-effects!

6. Remove duplicates

```python
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```python
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

```python
def test_???():
    return list(???(DUPLICATES))
```

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

```
def test_set():
    return list(set(DUPLICATES))
```

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

```
def test_set():
    return list(set(DUPLICATES))
```

6.07 ms (315/6.07 = 51)

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

```
def test_dict():
    # Works in CPython 3.6 and above
    return list(dict.fromkeys(DUPLICATES))
```

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

```
def test_dict():
    # Works in CPython 3.6 and above
    return list(dict.fromkeys(DUPLICATES))
```

11 ms (315/11 = 28.64)

6. Remove duplicates

```
# duplicates.py
from random import randrange
DUPLICATES = [randrange(100) for _ in range(1_000_000)]
```

```
def test_for_loop():
    unique = []
    for element in DUPLICATES:
        if element not in unique:
            unique.append(element)
    return unique
```

315 ms

```
def test_dict():
    # Works in CPython 3.6 and above
    return list(dict.fromkeys(DUPLICATES))
```

11 ms (315/11 = 28.64)

Only works with **hashable** keys!

Bonus: Different Python versions

```
# versions_benchmark.sh

# Ensure we don't write bytecode to __pycache__
export PYTHONDONTWRITEBYTECODE=1

echo "1. Permission vs. forgiveness"
echo "Permission 1 attribute:"
python -m timeit -s "from permission_vs_forgiveness import test_permission" "test_permission()"
echo "Forgiveness 1 attribute:"
python -m timeit -s "from permission_vs_forgiveness import test_forgiveness" "test_forgiveness()"

...

echo "\n6. Remove duplicates"
echo "For loop:"
python -m timeit -s "from duplicates import test_for_loop" "test_for_loop()"
echo "List comprehension:"
python -m timeit -s "from duplicates import test_list_comprehension" "test_list_comprehension()"
echo "Set:"
python -m timeit -s "from duplicates import test_set" "test_set()"
echo "Dict:"
python -m timeit -s "from duplicates import test_dict" "test_dict()"
```

Bonus: Different Python versions

```
$ pyenv shell 3.7.13

$ ./versions_benchmark.sh
1. Permission vs. forgiveness
Permission 1 attribute:
5000000 loops, best of 5: 58 nsec per loop
Forgiveness 1 attribute:
5000000 loops, best of 5: 41 nsec per loop
Permission 3 attributes:
2000000 loops, best of 5: 147 nsec per loop
Forgiveness 3 attributes:
...

$ pyenv shell 3.8.13
$ ./versions_benchmark.sh
...
```

What is pyenv and how to use it: https://switowski.com/blog/pyenv

	3.7.13	3.8.13	3.9.12	3.10.4	3.11.0b3	3.7 vs. 3.11
Permission (1 attr.)	89.7 ns	70.3 ns	71.3 ns	71.1 ns	**53.3 ns**	1.68
Forgiveness (1 attr.)	54 ns	48.6 ns	50.2 ns	56.2 ns	**33.9 ns**	1.59
Permission (3 attr.)	220 ns	**144 ns**	146 ns	150 ns	130 ns	1.69
Forgiveness (3 attr.)	90.8 ns	**69.6 ns**	72.4 ns	80.9 ns	60.2 ns	1.51
Permission (missing attr.)	116 ns	**84.7 ns**	85.1 ns	81.3 ns	**61.4 ns**	1.89
Forgiveness (missing attr.)	272 ns	264 ns	259 ns	305 ns	313 ns	0.87
Find element while loop	61 µs	61.9 µs	61.7 µs	59.1 µs	**48.1 µs**	1.27
Find element for loop	47 µs	47.3 µs	47.2 µs	46.5 µs	**40.8 µs**	1.15
Find element list comprehension	261 µs	263 µs	262 µs	252 µs	**217 µs**	1.20
Find element generator	47.1 µs	47.4 µs	47.6 µs	45.5 µs	**39.6 µs**	1.19
Filter list - loop	35.1 ms	34.5 ms	34.8 ms	33.5 ms	**26.8 ms**	1.31
Filter list - filter	47 ms	48.8 ms	51.9 ms	49.5 ms	**39.7 ms**	1.18
Filter list - comprehension	26.1 ms	26 ms	27.2 ms	25.6 ms	24.8 ms	1.05

Shannon Plan

markshannon / faster-cpython Public

Code Issues 3 Pull requests 1

master 1 branch 0 tags Go to file Add file Code

markshannon Merge pull request #2 from brettcannon/patch-1 ... 314eec8 on Oct 28, 2020 9 commits

README.md	Add files.	2 years ago
funding.md	Add files.	2 years ago
plan.md	Merge pull request #2 from brettcannon/patch-1	2 years ago
tiers.md	Add files.	2 years ago

README.md

A faster CPython

Five times faster

We want to speed up CPython by a factor of 5 over the next four releases.

See the plan for how this can be done.

The stages to high performance

Each stage will be 50% faster: 1.54 ≈ 5**

Stage 1 -- Python 3.10

The key improvement for 3.10 will be an adaptive, specializing interpreter. The interpreter will adapt to types and values during execution, exploiting type stability in the program, without needing runtime code generation.

Stage 2 -- Python 3.11

This stage will make many improvements to the runtime and key objects. Stage two will be characterized by lots of "tweaks", rather than any "headline" improvement. The planned improvements include:

- Improved performance for integers of less than one machine word.
- Improved peformance for binary operators.
- Faster calls and returns, through better handling of frames.
- Better object memory layout and reduced memory management overhead.
- Zero overhead exception handling.
- Further enhancements to the interpreter
- Other small enhancements.

Stage 3 -- Python 3.12 (requires runtime code generation)

Simple "JIT" compiler for small regions. Compile small regions of specialized code, using a relatively simple, fast compiler.

Stage 4 -- Python 3.13 (requires runtime code generation)

Extend regions for compilation. Enhance compiler to generate superior machine code.

	3.7.13	3.8.13	3.9.12	3.10.4	3.11.0b3	3.7 vs. 3.11
Membership* - for loop	6.58 ms	6.56 ms	6.31 ms	6.29 ms	**4.26 ms**	1.54
Membership* - in list	3.44 ms	3.42 ms	**2.99 ms**	3 ms	2.91 ms	1.18
Membership* - in set (cheating)	56.5 ns	54.6 ns	53.7 ns	51.5 ns	**35.1 ns**	1.61
Membership* - in set (proper)	10.8 ms	11.2 ms	11.3 ms	11.5 ms	11.8 ms	0.92
dict()	56.3 ns	59.1 ns	**46.2 ns**	**39.1 ns**	**28.7 ns**	1.96
{}	17.7 ns	**13.1 ns**	14.2 ns	14 ns	13.7 ns	1.29
Remove duplicates - for loop	363 ms	361 ms	**304 ms**	316 ms	314 ms	1.16
Remove duplicates - list comprehension	364 ms	357 ms	**307 ms**	317 ms	311 ms	1.17
Remove duplicates - set	5.55 ms	5.56 ms	5.78 ms	6.05 ms	6.11 ms	0.91
Remove duplicates - dict	9.49 ms	9.46 ms	11 ms	11 ms	10.8 ms	0.88

*Membership - checks number 500,000 in a 1,000,000 numbers list

	3.7.13	3.8.13	3.9.12	3.10.4	3.11.0a7	3.7 vs. 3.11
Membership* - for loop	6.58 ms	6.56 ms	6.31 ms	6.29 ms	4.3 ms	1.53
Membership* - in list	3.44 ms	3.42 ms	2.99 ms	3 ms	2.93 ms	1.17
Membership* - in set (cheating)	56.5 ns	54.6 ns	53.7 ns	51.5 ns	37.5 ns	1.51
Membership* - in set (proper)	10.8 ms	11.2 ms	11.3 ms	11.5 ms	11.5 ms	0.94
dict()	56.4 ns	59.1 ns	46.2 ns	39.1 ns	28.9 ns	1.95
{}	17.7 ns	13.1 ns	14.2 ns	14 ns	14 ns	1.26
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	1.20
Remove duplicates - list comprehension	364 ms	357 ms	307 ms	317 ms	304 ms	1.20
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted] ms	0.91
Remove duplicates - dict	9.49 ms	9.46 ms	11 ms	11 ms	10.7 ms	0.89
if variable == True	12.1 ns	11.7 ns	11.2 ns	11 ns	11.3 ns	1.07
if variable is True	8.4 ns	8.18 ns	8.22 ns	8.26 ns	8.45 ns	0.99
if variable	5.14 ns	4.97 ns	5.29 ns	6.19 ns	6.47 ns	0.79

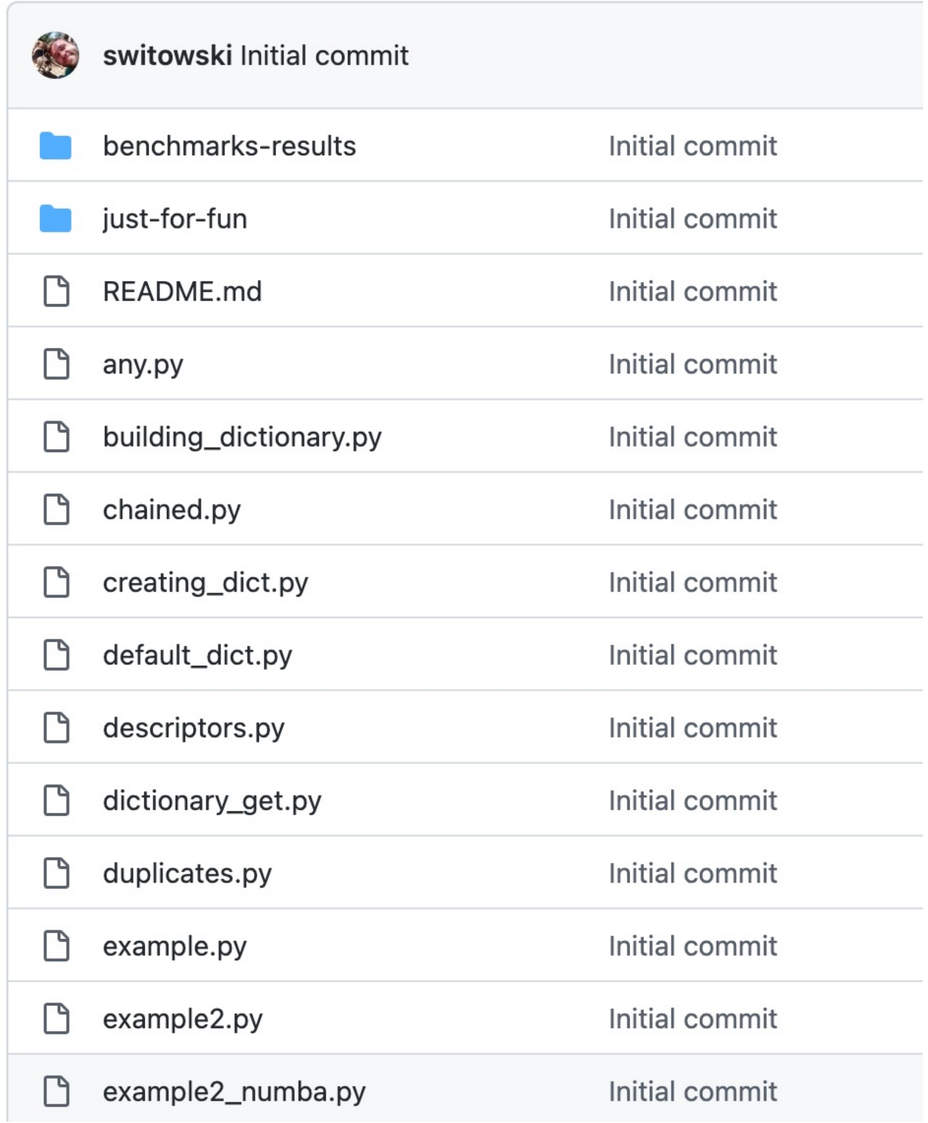
*Membership - checks number 500,000 in 1,000,000 numbers list

All the results are available in "*benchmarks-results*" folder in the repository:

https://github.com/switowski/writing-faster-python3

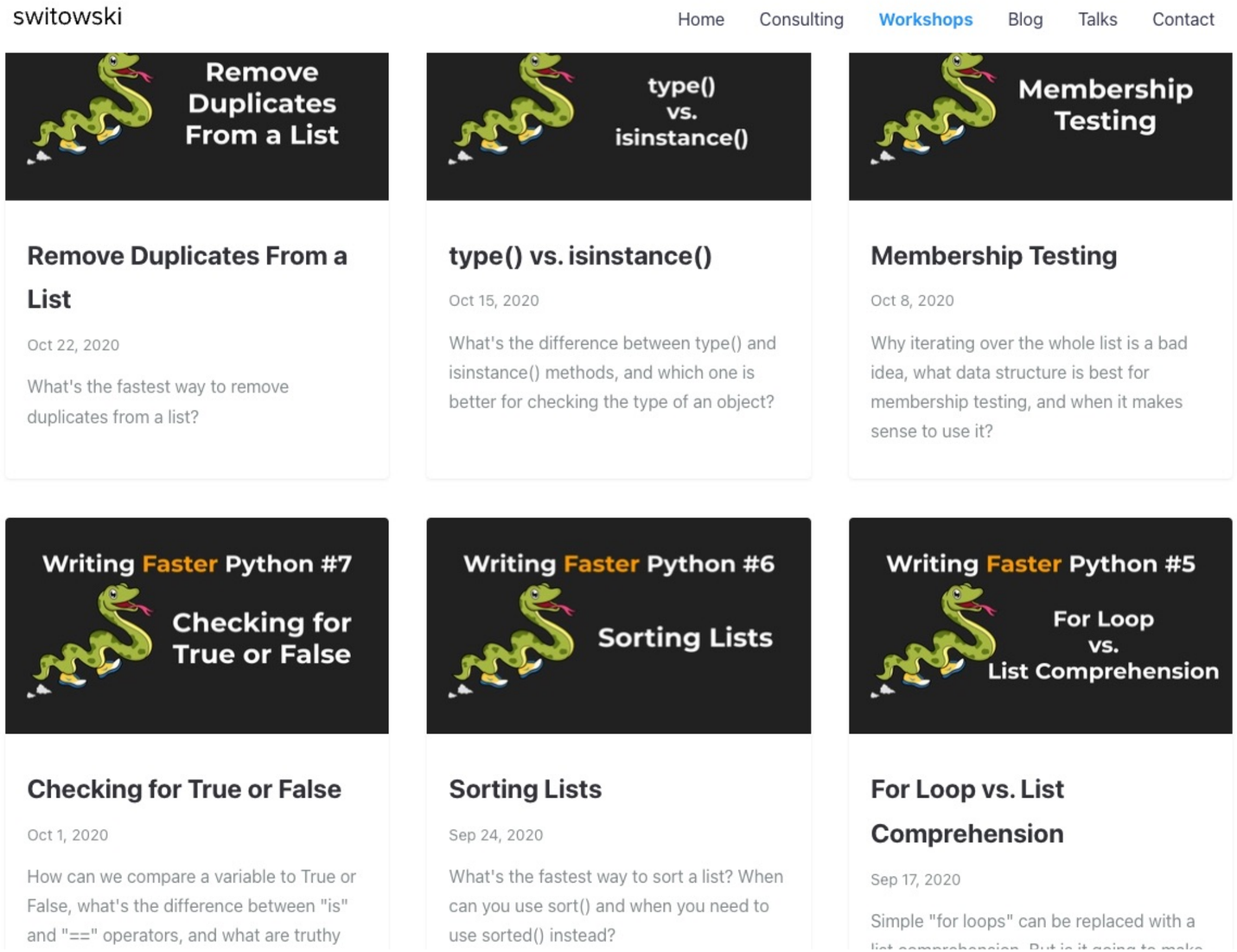
More examples

- For loop vs. dict comprehension
- dict[var] vs. dict.get(var)
- defaultdict vs. "manual default dict"
- ...



https://github.com/switowski/writing-faster-python3

Articles



https://switowski.com/tag/writing-faster-python

Conclusions

Source code optimization

- Source code optimization doesn't matter...

Source code optimization

- Source code optimization doesn't matter...
- ...except that it helps you write better Python code, use better data structures, and understand what your code does.

Source code optimization

- Source code optimization is cheap
- Source code optimization adds up
- Don't sacrifice readability for small performance gains

Thank you!

- Blog: https://switowski.com/blog
- "Writing Faster Python" series: https://switowski.com/tag/writing-faster-python
- GitHub repo: https://github.com/switowski/writing-faster-python3
- Slides: in the GitHub repo

Questions?

- Blog: https://switowski.com/blog
- "Writing Faster Python" series: https://switowski.com/tag/writing-faster-python
- GitHub repo: https://github.com/switowski/writing-faster-python3
- Slides: in the GitHub repo