

Writing Faster Python 3

Sebastian Witowski

Writing Faster Python 3



Revised version of "Writing Faster Python" from 2016.

- Python 3
- No theory (we all know that you should measure before you optimize)

```
# 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)
```

```
# 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
```

```
# 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
```




europython
July 8-14 2019
BASEL

Wait, IPython can do that?!

Sebastian Witowski

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


```
# 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
```

```
# 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
```



```
# 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)

```
# 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)

```
# 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)

```
# 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)


```
# 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
```

```
# 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)
```



```
# 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)
```

```
# 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)
```

```
# 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
```



```
# 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

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
```

```
F00 = Foo()
```

```
F00.hello
```

```
# permission_vs_forgiveness.py
```

```
class BaseClass:  
    hello = "world"
```

```
class Foo(BaseClass):  
    pass
```

```
F00 = Foo()
```

```
# Ask for permission
```

```
def test_permission():  
    if hasattr(F00, "hello"):  
        F00.hello
```

```
# Ask for forgiveness
```

```
def test_forgiveness():  
    try:  
        F00.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

```
# permission_vs_forgiveness2.py
```

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

```
class Foo(BaseClass):
    pass
```

```
F00 = Foo()
```

```
# Ask for permission
```

```
def test_permission():
    if hasattr(F00, "hello") and hasattr(F00, "bar") and hasattr(F00, "baz"):
        F00.hello
        F00.bar
        F00.baz
```

```
# Ask for forgiveness
```

```
def test_forgiveness():
    try:
        F00.hello
        F00.bar
        F00.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

```
# permission_vs_forgiveness3.py
```

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

```
class Foo(BaseClass):
    pass
```

```
F00 = Foo()
```

```
# Ask for permission
```

```
def test_permission():
    if hasattr(F00, "hello") and hasattr(F00, "bar") and hasattr(F00, "baz"):
        F00.hello
        F00.bar
        F00.baz
```

```
# Ask for forgiveness
```

```
def test_forgiveness():
    try:
        F00.hello
        F00.bar
        F00.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

```
# 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

```
# 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
```

```
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
```

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)

Ok, but let's try without cheating this time

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

```
# 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
```

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

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_???:
    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:
500000 loops, best of 5: 58 nsec per loop
Forgiveness 1 attribute:
500000 loops, best of 5: 41 nsec per loop
Permission 3 attributes:
200000 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	3.11.0a7	3.7 vs. 3.11
Permission (1 attr.)	89.7 ns	70.3 ns	71.3 ns	71.1 ns	58 ns	1.54
Forgiveness (1 attr.)	54 ns	48.6 ns	50.2 ns	56.2 ns	38.2 ns	1.41
Permission (3 attr.)	220 ns	144 ns	146 ns	150 ns	147 ns	1.50
Forgiveness (3 attr.)	90.8 ns	69.6 ns	72.4 ns	80.9 ns	76.2 ns	1.19
Permission (missing attr.)	116 ns	84.7 ns	85.1 ns	81.3 ns	65.4 ns	1.77
Forgiveness (missing attr.)	272 ns	264 ns	259 ns	305 ns	310 ns	0.88
Find element while loop	61 µs	61.9 µs	61.7 µs	59.1 µs	47.5 µs	1.28
Find element for loop	47 µs	47.3 µs	47.2 µs	46.5 µs	41.2 µs	1.14
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	39.1 µs	1.20
Filter list - loop	35.1 ms	34.5 ms	34.8 ms	33.5 ms	27.6 ms	1.27
Filter list - filter	47 ms	48.8 ms	51.9 ms	49.5 ms	40.2 ms	1.17
Filter list - comprehension	26.1 ms	26 ms	27.2 ms	25.6 ms	25.3 ms	1.03

Shannon Plan

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:
 $1.5^{4} \approx 5$**

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 performance 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	3.11.0a7	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()	56.3 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
Remove duplicates - for loop	363 ms	361 ms	304 ms	316 ms	302 ms	1.20
Remove duplicates - list comprehension	364 ms	357 ms	307 ms	317 ms	304 ms	1.20
Remove duplicates - set	5.55 ms	5.56 ms	5.78 ms	6.05 ms	6.07 ms	0.91
Remove duplicates - dict	9.49 ms	9.46 ms	11 ms	11 ms	10.7 ms	0.89

*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()	56.3 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
Remove duplicates - list comprehension	364 ms	357 ms	307 ms	317 ms	304 ms	1.20
Remove duplicates - set	5.59 ms	5.56 ms	5.77 ms	6.27 ms	2.62 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
















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

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

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

More examples

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

	switowski Initial commit	
	benchmarks-results	Initial commit
	just-for-fun	Initial commit
	README.md	Initial commit
	any.py	Initial commit
	building_dictionary.py	Initial commit
	chained.py	Initial commit
	creating_dict.py	Initial commit
	default_dict.py	Initial commit
	descriptors.py	Initial commit
	dictionary_get.py	Initial commit
	duplicates.py	Initial commit
	example.py	Initial commit
	example2.py	Initial commit
	example2_numba.py	Initial commit

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

Articles



Remove Duplicates From a List

Remove Duplicates From a List

Oct 22, 2020

What's the fastest way to remove duplicates from a list?



type() vs. isinstance()

type() vs. isinstance()

Oct 15, 2020

What's the difference between type() and isinstance() methods, and which one is better for checking the type of an object?



Membership Testing

Membership Testing

Oct 8, 2020

Why iterating over the whole list is a bad idea, what data structure is best for membership testing, and when it makes sense to use it?



Writing Faster Python #7

Checking for True or False

Oct 1, 2020

How can we compare a variable to True or False, what's the difference between "is" and "==" operators, and what are truthy



Writing Faster Python #6

Sorting Lists

Sep 24, 2020

What's the fastest way to sort a list? When can you use sort() and when you need to use sorted() instead?



Writing Faster Python #5

For Loop vs. List Comprehension

Sep 17, 2020

Simple "for loops" can be replaced with a list comprehension. But is it going to make

<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