

# HIGH-PERFORMANCE Python

## PART 1: BASIC TOOLS



# High-performance Python

<https://github.com/squillero/python-lectures/>

**Copyright © 2023 by Giovanni Squillero.**

Permission to make digital or hard copies for personal or classroom use of these files, either with or without modification, is granted without fee provided that copies are not distributed for profit, and that copies preserve the copyright notice and the full reference to the source repository. To republish, to post on servers, or to redistribute to lists, contact the Author. These files are offered as-is, without any warranty.

september 2023

giovanni.squillero@polito.it

version 0.5

High-performance Python

3

## Why HP Python?

- Python is not quite fast, at least not compared to C++ or Rust
- ... but it's not quite slow either
  - Large projects
  - System language
  - High-level algorithms
- Goal of HP Python
  - Avoid pitfalls
  - High-level optimization
  - Pure-Python optimization

giovanni.squillero@polito.it

High-performance Python



4

## Rule of Optimization (personal)

- Don't write it
- Don't do it
- Exploit parallelism

giovanni.squillero@polito.it

High-performance Python



5

## Rule of Optimization (personal)

- Don't write it
  - Use builtins
  - Use standard libraries
- Don't do it
- Exploit parallelism



giovanni.squillero@polito.it

High-performance Python



6

## Big O notation

- [https://en.wikipedia.org/wiki/Big\\_O\\_notation](https://en.wikipedia.org/wiki/Big_O_notation)



giovanni.squillero@polito.it

High-performance Python

7



PREMATURE  
OPTIMIZATION  
IS THE ROOT  
OF ALL EVIL



1. MAKE IT RUN

## Rules of Thumb

1. Write as few lines of code as possible
  1. Use builtins / standard library
  2. Use generators

giovanni.squillero@polito.it

High-performance Python

10

## Good Practice: Virtual Environments

- venv
- poetry
- ...

giovanni.squillero@polito.it

High-performance Python

11

## Problem

- Given  $n$
- Find the *first* Collatz sequence including it  
(i.e., the sequence starting with the smallest positive integer)

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2}, \\ 3n + 1 & \text{if } n \equiv 1 \pmod{2}. \end{cases}$$

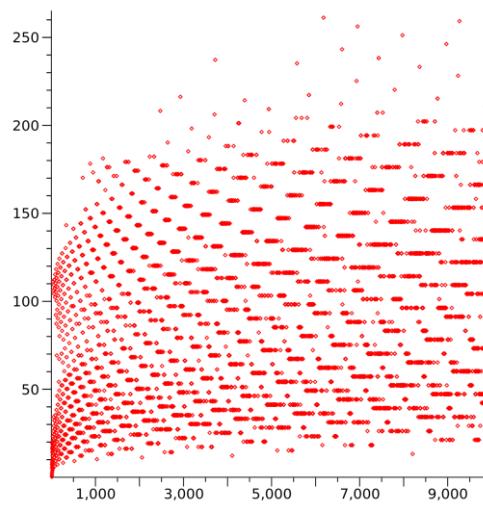
1	1
2	2, 1
3	3, 10, 5, 16, 8, 4, 2, 1
4	4, 2, 1
5	5, 16, 8, 4, 2, 1
6	6, 3, 10, 5, 16, 8, 4, 2, 1

giovanni.squillero@polito.it

High-performance Python

12

## Collatz 1-9999



giovanni.squillero@polito.it

High-performance Python

13



## Good Practice: assert

```
def division(num1: int, num2: int):
    assert num2!= 0, "num2 must be different from 0"
    return num1/num2
```

## Good Practice: pytest



# pytest

## pytest (Basic Usage)

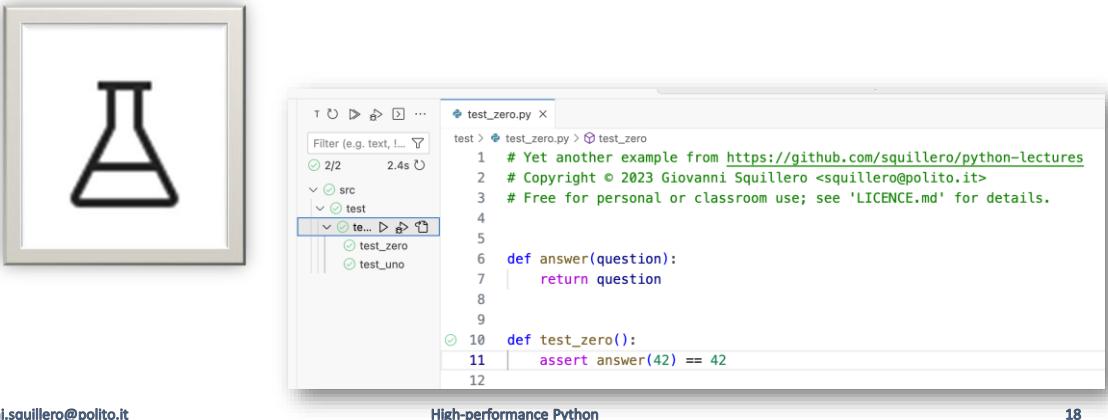
- Checks values through asserts
- Directory ‘test’
- Filenames start with ‘test’

```
def answer(question):
    return question

def test_zero():
    assert answer(42) == 42
```

## Basic Usage

- Integrated in Visual Studio Code and PyCharm



```
test > test_zero.py > test_zero
1 # Yet another example from https://github.com/squillero/python-lectures
2 # Copyright © 2023 Giovanni Squillero <squillero@polito.it>
3 # Free for personal or classroom use; see 'LICENCE.md' for details.
4
5
6 def answer(question):
7     return question
8
9
10 def test_zero():
11     assert answer(42) == 42
12
```

giovanni.squillero@polito.it

18

## Advanced Usage

- Extended checking (warnings, etc.)
- Fixture
- Parametrized checks
- Optional checks
- ...

```
@pytest.mark.avoidable
@pytest.mark.parametrize('generator,size', itertools.product(generators, test_sizes))
def test_individual_creation(individuals, generator, size):
    bar1 = byron.f.macro('bar {ref}', ref=byron.f.global_reference('bunch1'))
    bar2 = byron.f.macro('bar {ref}', ref=byron.f.global_reference('bunch2', first_macro=True))
    bunch1 = byron.f.bunch(get_base_macros() + [bar2], size=size // 2, name='bunch1')
    bunch2 = byron.f.bunch(get_base_macros() + [bar1], size=size // 2, name='bunch2')
    body = byron.f.sequence([bunch1, bunch2])
    individuals[size].extend(generator(body))
    individuals[size].extend(generator(body))
```

giovanni.squillero@polito.it

High-performance Python

19



## Timing

- Use Magic

```
%timeit foo(x)
%timeit
```

```
In [1]: %timeit
        dict(type = 'grid',
             name='XGBoost Grid Search',
             parameters={
                 'min_child_weight': bounds=dict(min=1, max=10), type='int', grid=min_child_weight_grid,
                 'max_depth': bounds=dict(min=1, max=11), type='int', grid=max_depth_grid,
                 'n_estimators': bounds=dict(min=1, max=256), type='int', grid=n_estimators_grid},
                 dict(name='log_learning_rate', bounds=dict(min=np.log(0.1), max=np.log(1)), type='double', grid=log_learning_rate_grid),
                 )
             metrics={
                 dict(name='AUCROC', objective='maximize', strategy='optimize'),
                 dict(name='F1 Score', strategy='stocic')
             }
         )

In [1]: %timeit
        xgopt.log_dataset(
            fraud_fullset,
            type='vector',
            version='v1',
            features=[f'mean{trainX}', f'mean{textX}', f'mean{testX}'],
            f'mean{text}', f'mean{testX}'],
            f'features={features}'
        ),
        xgopt.log_metadata('len(trainX)', len(trainX))
        xgopt.log_metadata('len(test)', len(testX))
        xgopt.log_metadata('features', ', '.join(features))
        xgopt.log_model('xgbmodel_sklearn.XGBModelClassifier')
    #model parametrization
```

# Timing

- **timeit**
  - Simple way to time small bits of Python code
  - Avoids a number of “common traps” for measuring execution times

```
timeit.timeit(lambda: shell_task(1), number=1)
1.028357582999888

timeit.repeat(lambda: shell_task(1), repeat=5, number=2)
[2.052590582999983,
 2.045524707998993,
 2.043365708002966,
 2.0421462500016787,
 2.0401462910012924]
```

giovanni.squillero@polito.it

High-performance Python

22

# Timing

- **pytest-performance**
  - “A simple plugin to ensure the execution of critical sections of code has not been impacted”

giovanni.squillero@polito.it

High-performance Python

23

# Profiling

- Default profile + Snakeviz

```
python -O -m cProfile -o foo.prof ./foo.py
snakeviz foo.prof
```

giovanni.squillero@polito.it

High-performance Python

24

# Profiling in Notebooks

```
%load_ext snakeviz
from time import sleep

def foo(x):
    print(f"Sleeping {x} ms")
    sleep(x/1000)

def bar(x):
    for i in range(x):
        foo(i+1)

%snakeviz bar(5)
```

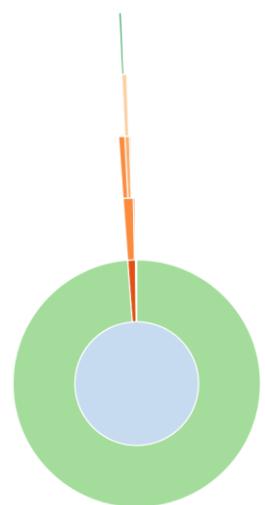
Reset Zoom

Style: Sunburst ▾

Depth: 10 ▾

Cutoff: 1 / 1000 ▾

Name:  
<built-in  
method  
time.sleep>  
Cumulative  
Time:  
0.0186 s  
(98.43 %)  
File:  
~  
Line:  
0  
Directory:

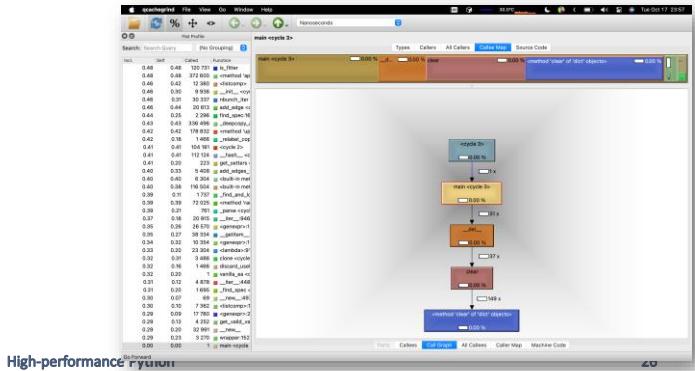


giovanni.squillero@polito.it

## Profiling

- KCachegrin (QCacheGrin on MacOS)  
pyprof2calltree -i foo.prof  
qcachegrind foo.prof.log

[giovanni.squillero@polito.it](mailto:giovanni.squillero@polito.it)



**High-performance** Go Forward. **Ryuum**

## Even More Profiling

- `line_profile`
  - `memory_profile`
  - <https://godbolt.org/>

[giovanni.squillero@polito.it](mailto:giovanni.squillero@polito.it)

High-performance Python

# Fundamental Algorithms & Data Structures



giovanni.squillero@polito.it

High-performance Python

28

## Lists and Tuples

- Lists
  - Dynamic arrays (mutable)
  - Size vs. Space
- Tuples
  - Static arrays (immutable)
  - Can be cached by the Python runtime
- Complexity:
  - Most operations are  $O(1)$  + resizing

giovanni.squillero@polito.it

High-performance Python

29

# Dictionaries and Sets

- Dictionaries and sets are nearly identical
  - A dictionaries maps unique keys to values
  - Dictionaries remember insertion order (negligible overhead)
  - A set can be defined as a collection of keys (no values)
  - Sets are very useful for doing set operations (D'ho!?)
- Keys need to be *hashable*
  - Hashable objects need to be immutable
- Most operations are like O(1)

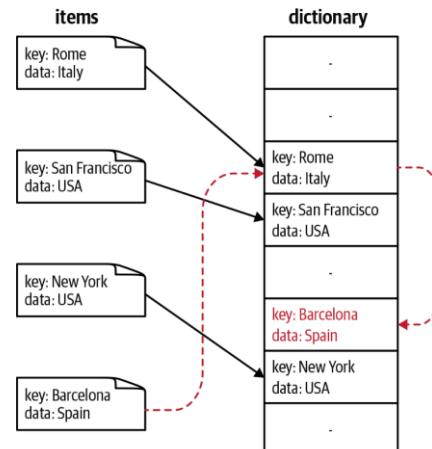
giovanni.squillero@polito.it

High-performance Python

30

# Under the hood

- Collisions
- Insertion
- Deletion
- Resizing



giovanni.squillero@polito.it

High-performance Python

31

# Namespaces

- Namespaces are dictionaries!
- **`globals()`**
  - The dictionary implementing the current module namespace
- **`locals()`**
  - The dictionary representing the current local symbol table
- **Notes:**
  1. At the module level, `locals` and `globals` are the same dictionary
  2. The contents of these dictionaries should not be modified

# Deque

- List
  - `pop/append` are  $O(1)$
  - `pop(0)/insert(0, ★)` are  $O(N)$
  - Access elements is  $O(1)$
- Deque
  - `pop/popleft & append/appendright` are  $O(1)$
  - Access elements in the middle is like  $O(N)$

Notez bien: not considering resizing

## Queues

- Multi-producer, multi-consumer queues (thread safe, but also useful in non-threaded environment)
- Standard queues (put, get, full, empty)
  - Queue
  - SimpleQueue
  - LifoQueue
  - PriorityQueue

## Priority Queues

- Use tuples
- Use custom classes
  - With `@functools.total_ordering`, a class needs only to define `__eq__()` and one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`

# Caching and Memoization

`@functools.cache`

`@functools.cached_property`

`@functools.lru_cache(maxsize=128, typed=False)`

giovanni.squillero@polito.it

High-performance Python

36



giovanni.squillero@polito.it