HIGH-PERFORMANCE Python

PART 1: BASIC TOOLS

High-performance Python

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

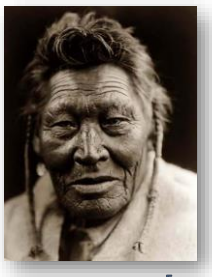
september 2023 **version 0.5**

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

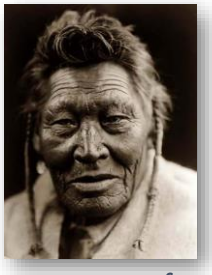
Rule of Optimization (personal)

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



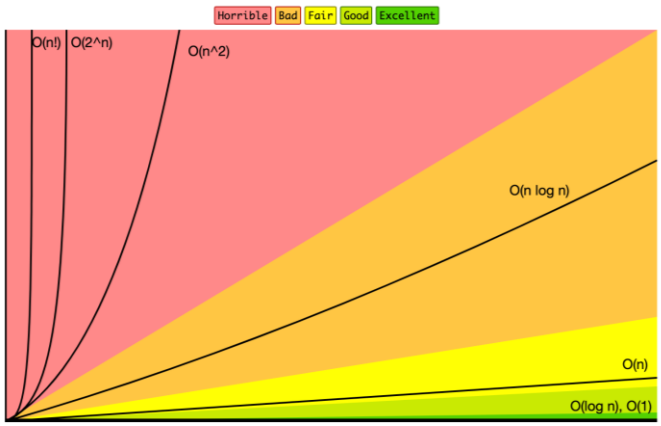
Rule of Optimization (personal)

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



Big *O* notation

- https://en.wikipedia.org/wiki/Big_O_notation




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

Good Practice: Virtual Environments

- `venv`
- `poetry`
- …

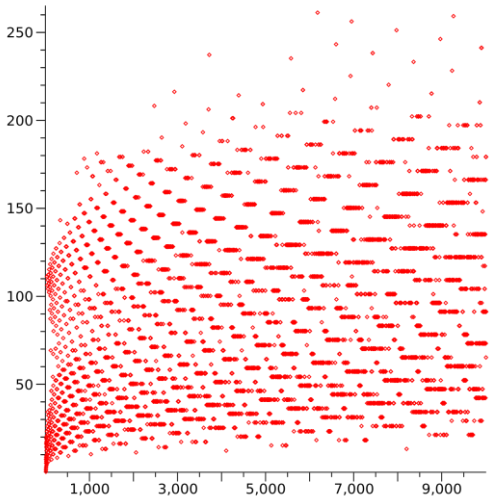
Problem

- 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

Collatz 1-9999



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 (Basic Usage)

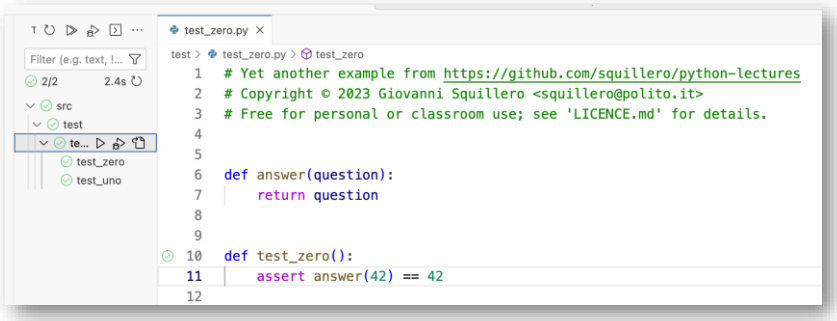
- Checks values through asserts
- Directory 'test'
- Filenames start with 'test'

```python
def answer(question):
    return question


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

Basic Usage

- Integrated in Visual Studio Code and PyCharm



Advanced Usage

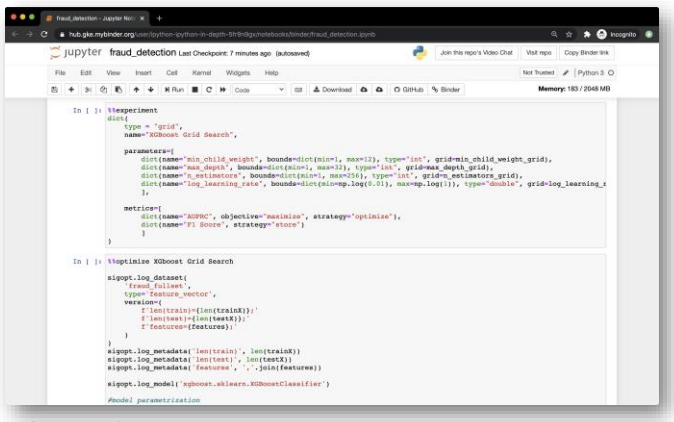
- Extended checking (warnings, etc.)
- Fixture
- Parametrized checks
- Optional checks
- …

```
@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))
```

Timing

- Use Magic

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



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]
```


Timing

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

Profiling

- Default profile + Snakeviz

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


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)
```

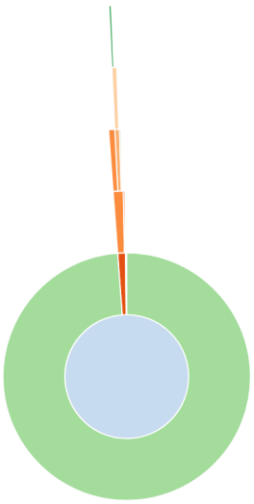
Reset Zoom

Style: Sunburst

Depth: 10

Cutoff: 1 / 1000

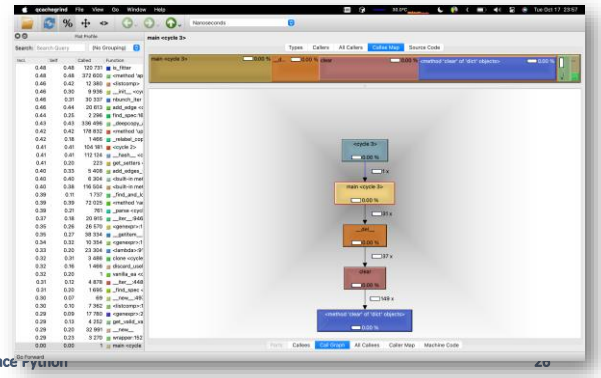
Name:
<built-in
method
time.sleep>
Cumulative
Time:
0.0186 s
(98.43 %)
File:
~
Line:
0
Directory:



Profiling

- KCachegrin (QCacheGrin on MacOS)

```
pyprof2calltree -i foo.prof
qcachegrind foo.prof.log
```



Even More Profiling

- `line_profile`
- `memory_profile`

- https://godbolt.org/

Fundamental Algorithms & Data Structures

Lists and Tuples

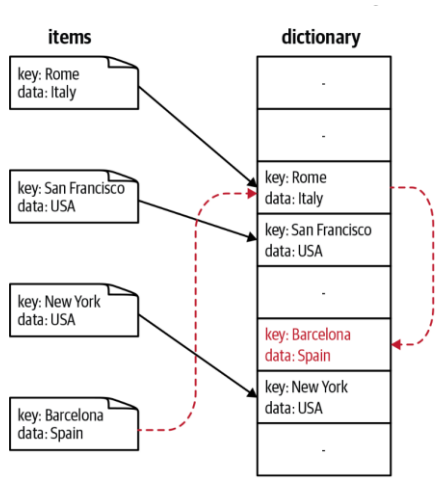
- Lists
  - Dynamic arrays (mutable)
  - Size vs. Space
- Tuples
  - Static arrays (immutable)
  - Can be cached by the Python runtime
- Complexity:
  - Most operations are O(1) + resizing

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)

Under the hood

- Collisions
- Insertion
- Deletion
- Resizing



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)
```