High-performance Python

https://staff.polito.it/giovanni.squillero/lectures/

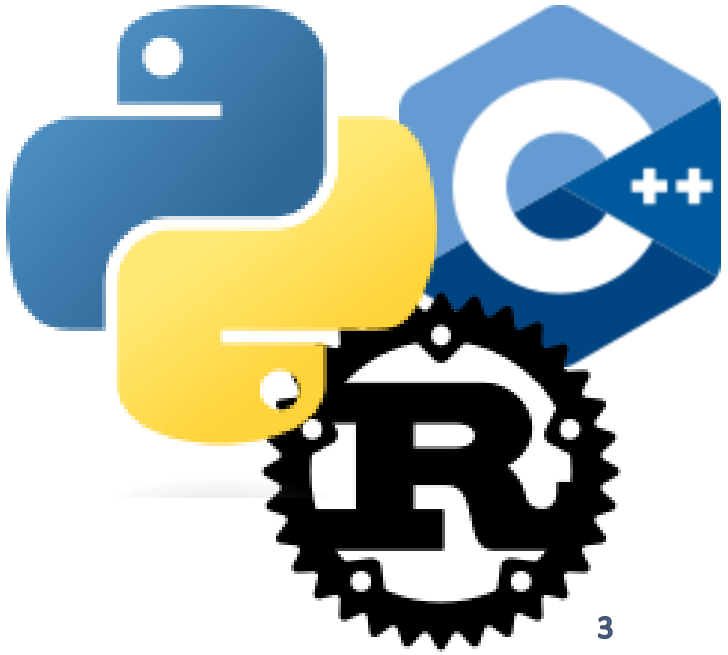
september 2023 version 1.0

High-performance Python

Part 1: Basic Tools

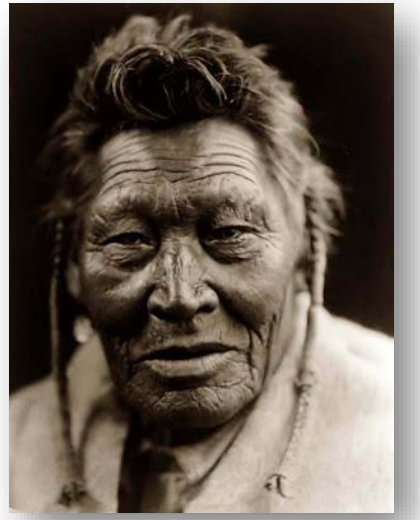
Why HP Python?

- 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



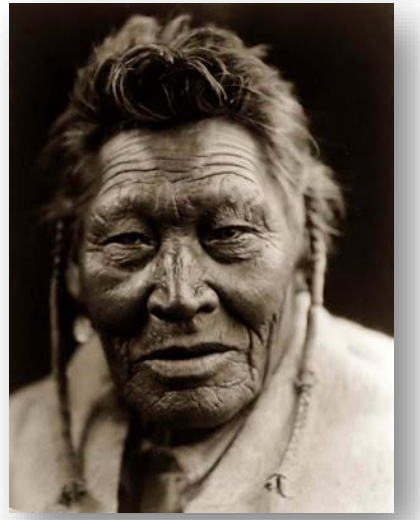
Rule of Optimization (personal)

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



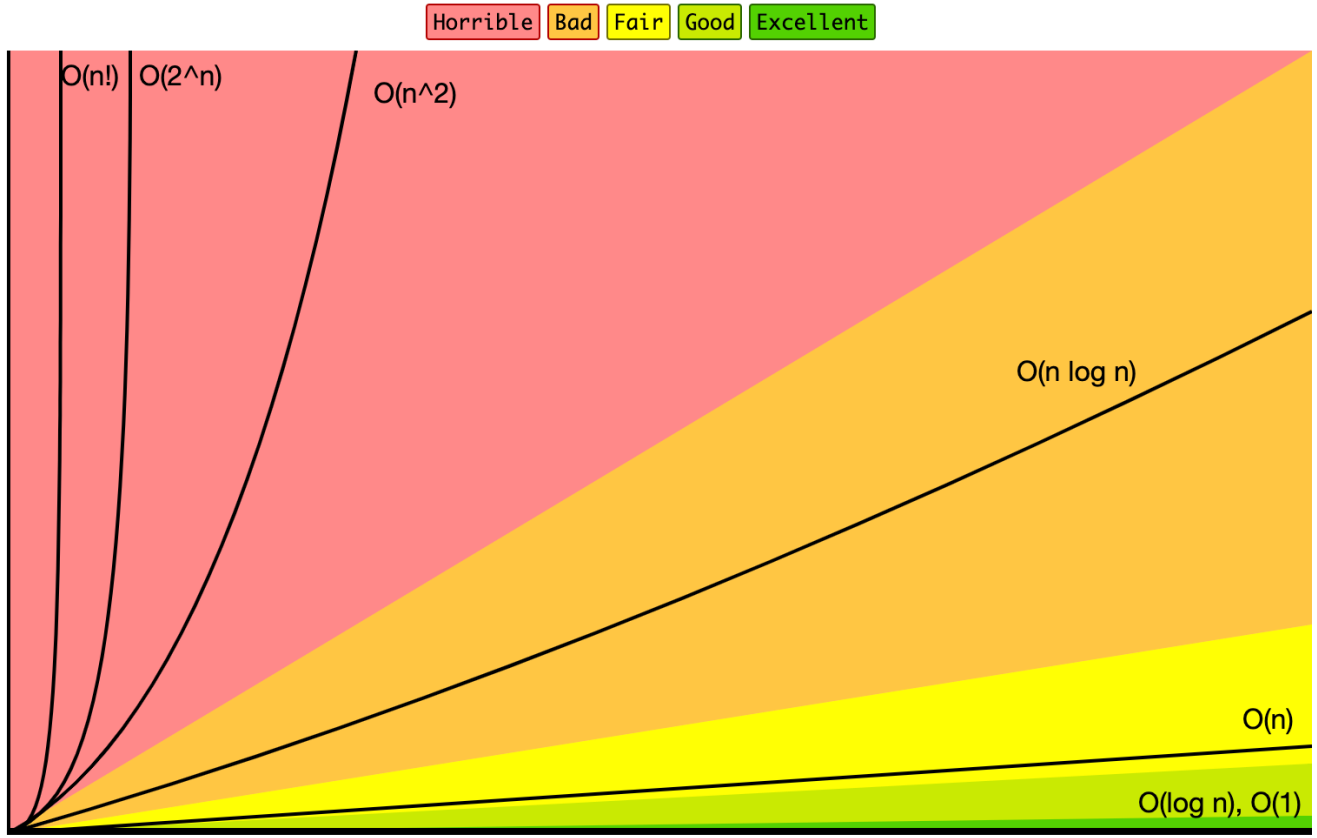
Rule of Optimization (personal)

- 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`
- …

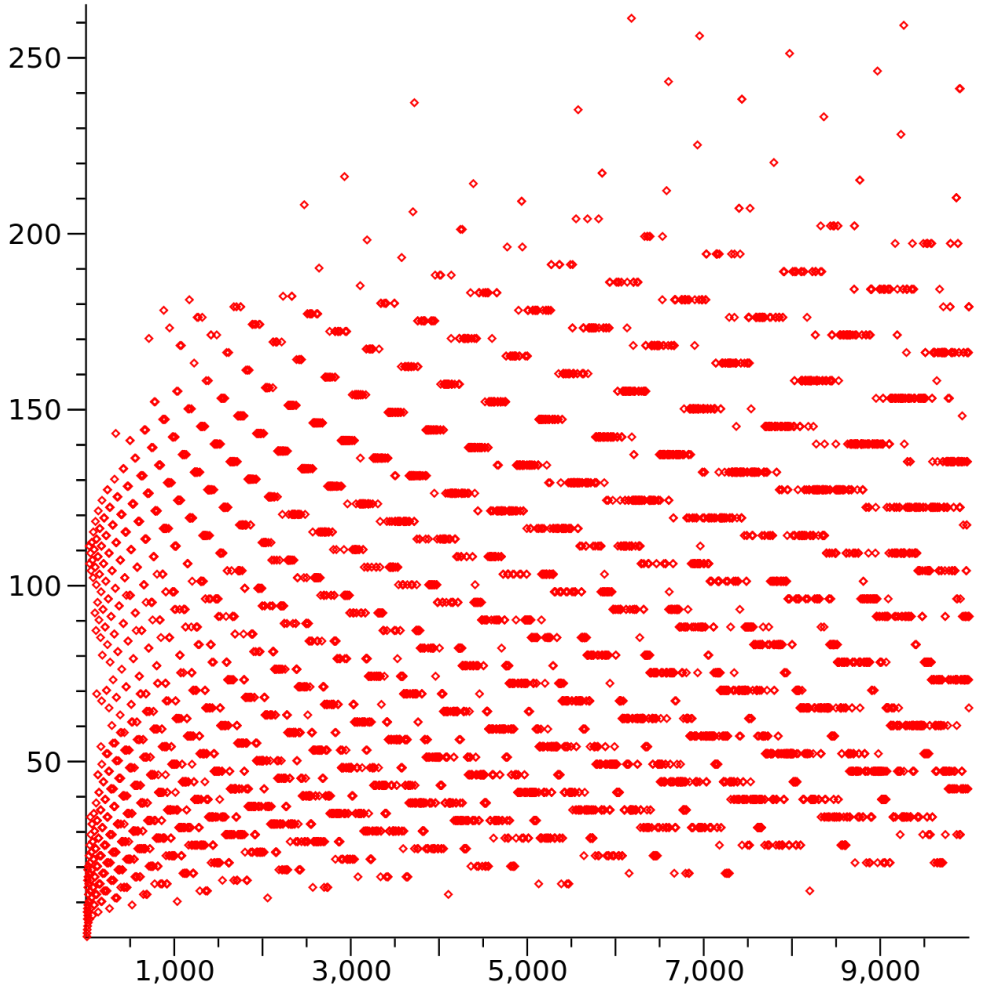
Problem

- 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




1. Make it RUN
2. Make it RIGHT

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)

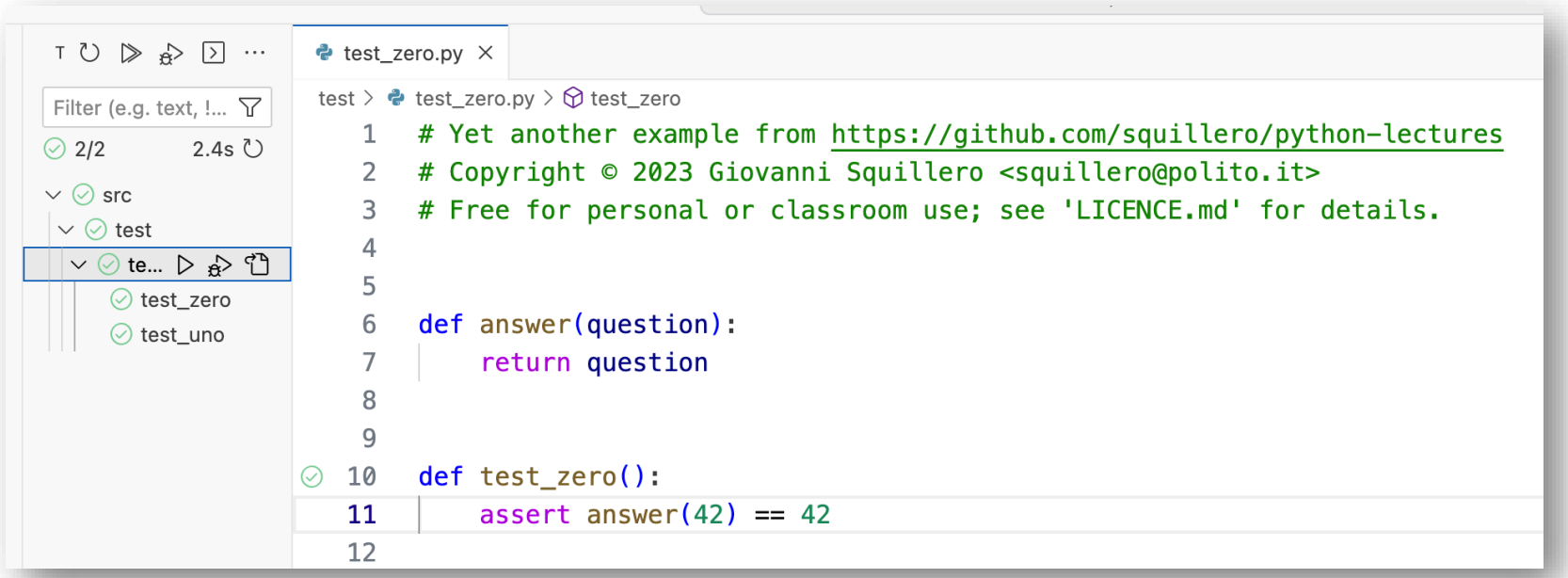
- Checks values through asserts
- Directory ‘test’
- Filenames start with ‘test’

```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
- …

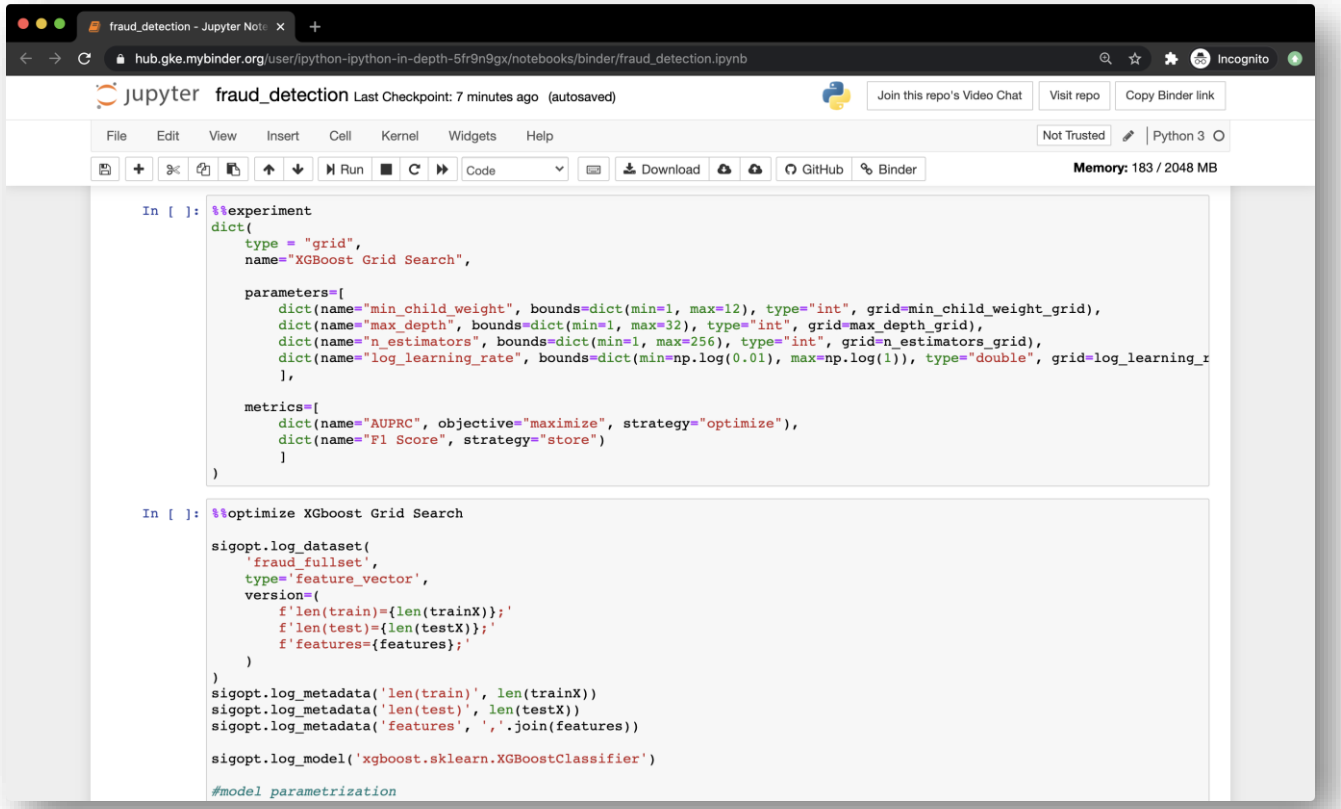
```python
@pytest.mark.avoidable
@pytest.mark.parametrize('generator,size', itertools.product(generators, test_sizes))
def test_individual_creation(individuals, generator, size):
    bar1 = byron.f.macro('bar {ref}', ref=byron.f.global_reference('bunch1'))
    bar2 = byron.f.macro('bar {ref}', ref=byron.f.global_reference('bunch2', 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))
```


1. Make it RUN
2. Make it RIGHT
3. Make it FAST

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)
```

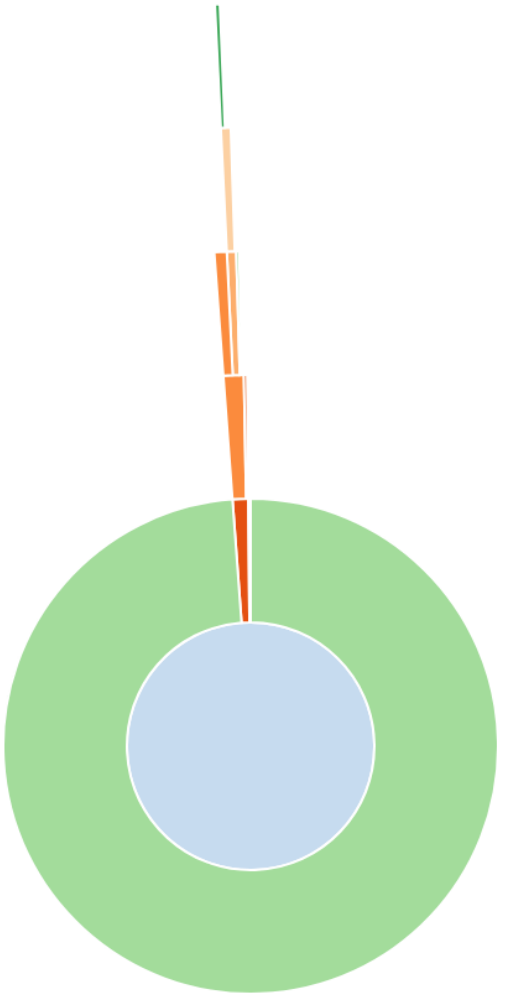
Reset Zoom

Style: Sunburst

Depth: 10

Cutoff: 1 / 1000

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

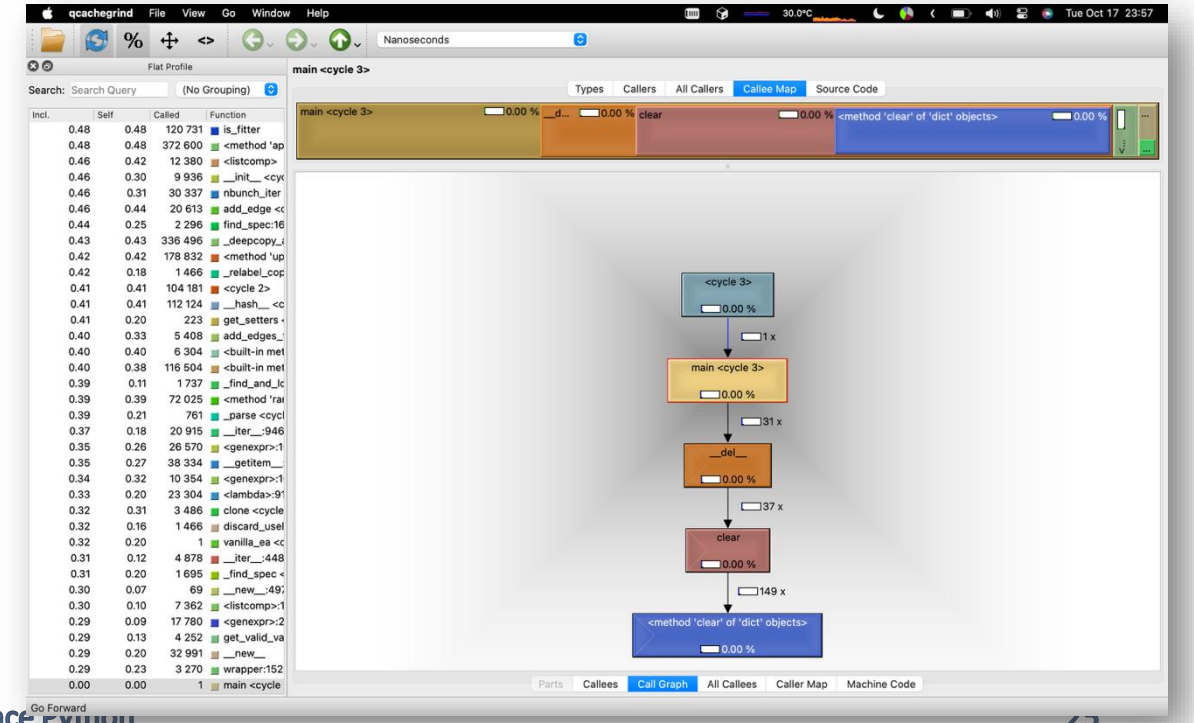


giovanni.squillero@polito.it

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

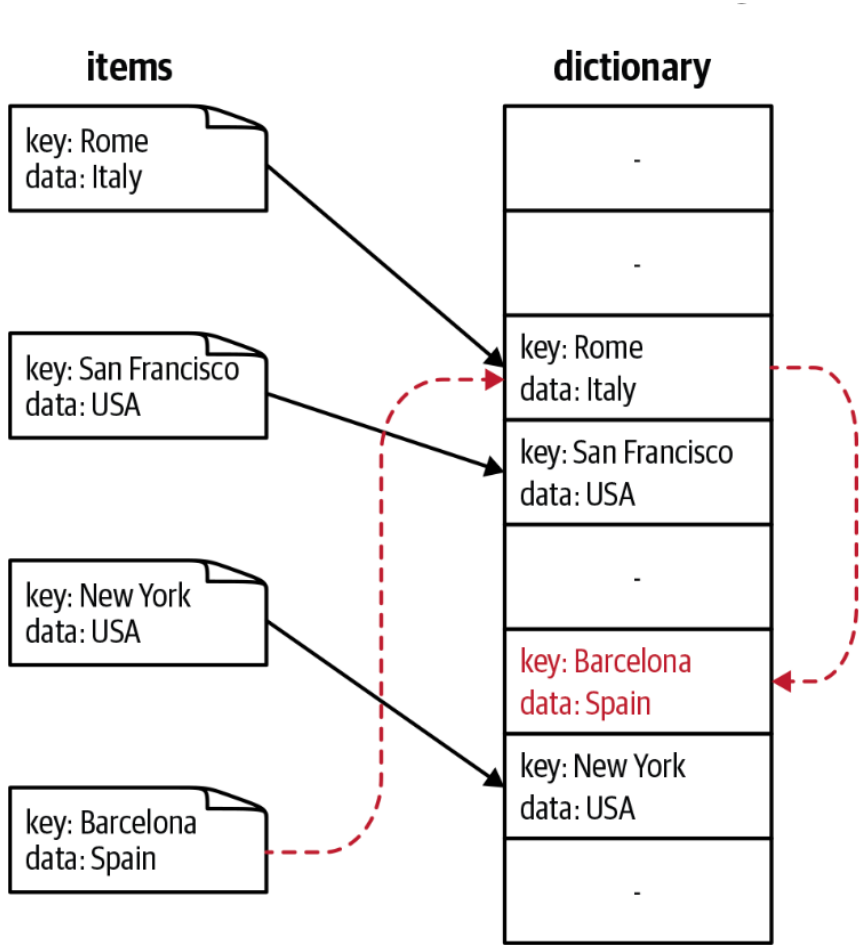
- Lists
  - Dynamic arrays (mutable)
  - Size vs. Space
- Tuples
  - Static arrays (immutable)
  - Can be cached by the Python runtime
- Complexity:
  - Most operations are O(1) + resizing

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

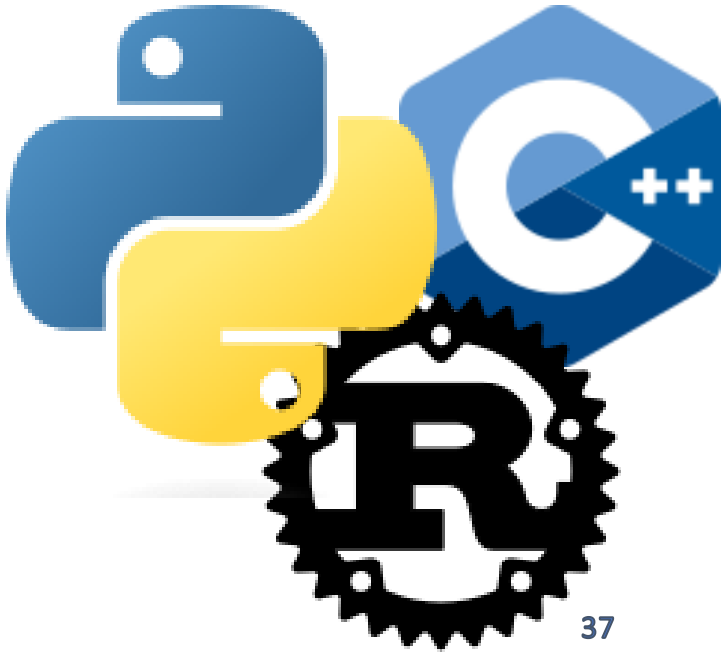
```
@functools.cache
@functools.cached_property
@functools.lru_cache(maxsize=128, typed=False)
```

High-performance Python

Part 2: Single thread

Why HP Python?

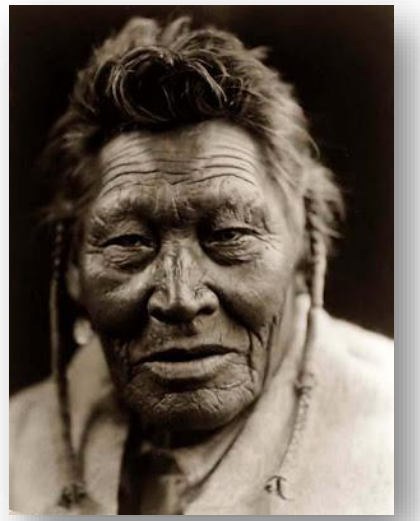
- Python is not quite fast, at least not compared to C++ or Rust
- … but it's not quite slow either
  - Large projects
  - System language
  - High-level algorithms
- Goal of HP Python
  - Avoid pitfalls
  - High-level optimization
  - Pure-Python optimization



Rule of Optimization (personal)

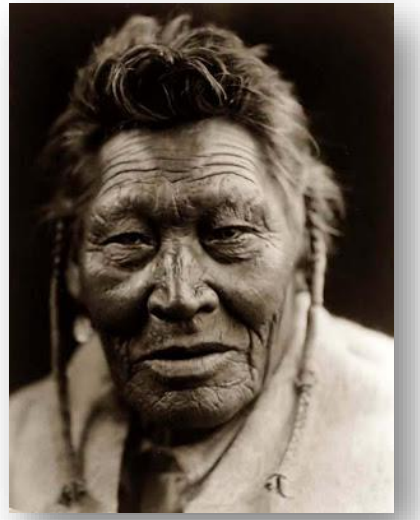
- Don’t write it
  - Use builtins
  - Use standard libraries
- Don’t do it
- Exploit parallel

Lecture 1



Rule of Optimization (personal)

- Don’t write it
  - Use builtins
  - Use standar
- Don’t do it
  - Don’t do it yet: i.e., use lazy execution
  - Don’t do it again: i.e., cache previous calls
- Exploit parallelism



Lazy Executions & Generators

```
def number_generator(n):
    i = 0
    while i < n:
        yield i
        i += 1

for number in number_generator(5):
    print(number)
```

Problem: Write a Generator for Fibonacci

Generators & Lazy Evaluation

- `any()`
- `all()`

Problem: Calculate Prime Numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

DON'T DO IT

Out-of-the-box


```
import functools
```

```
@functools.cache
def cached_function(num):
    print(f"Performing costly calculation for {num}")
    return num * 2
```

```
cached_function(42)
```

```
Performing costly calculation for 42
84
```

```
cached_function(42)
```

```
84
```

```
cached_function(43)
```

```
Performing costly calculation for 43
86
```

Out-of-the-box

```
@functools.lru_cache(maxsize=512)
def cached_function(num):
    print(f"Performing costly calculation for {num}")
    return num * 2
```

```
cached_function(42)
```

```
Performing costly calculation for 42
84
```

```
cached_function.cache_info()
```

```
CacheInfo(hits=0, misses=1, maxsize=128, currsize=1)
```

```
cached_function.cache_clear()
```

```
cached_function(42)
```

```
Performing costly calculation for 42
84
```

Out-of-the-box (caveats)

- `@functools.cached_property`
  - Similar to property, but with caching
- `@functools.lru_cache(typed)`
  - If `typed` is `True`, function arguments of different types will be cached separately
  - If `typed` is `False` (default), the implementation will *"usually"* regard different types as equivalent calls and only cache a single result — but some types *"may"* be always cached separately

Memoization: joblib

```
from joblib import Memory
memory = Memory('./cache', verbose=0)
```

```
@memory.cache
def memoized_function(num):
    print(f"Performing even more costly calculation for {num}")
    return num * 3
```

```
memoized_function(13)
```

```
Performing even more costly calculation for 13
39
```

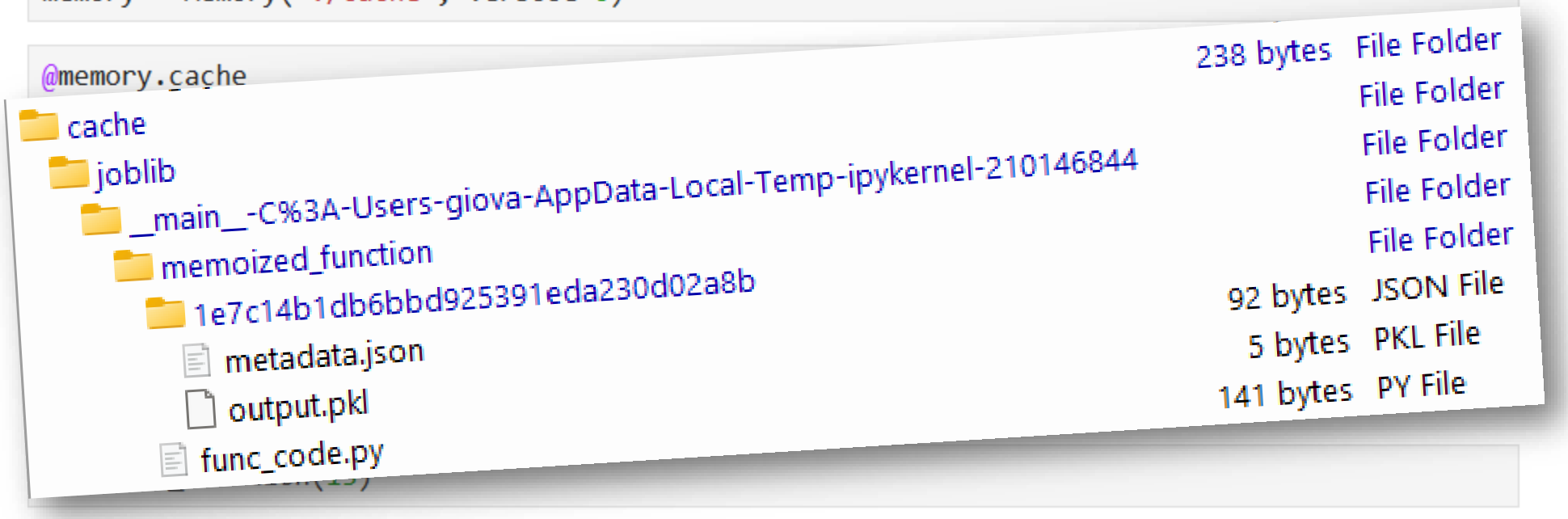
```
memoized_function(13)
```

```
39
```

Memoization: joblib

```python
from joblib import Memory
memory = Memory('./cache', verbose=0)

@memory.cache
```




```
39
```

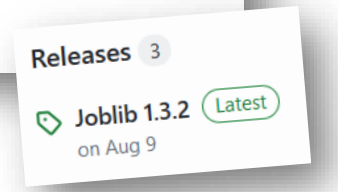
Memoization: joblib

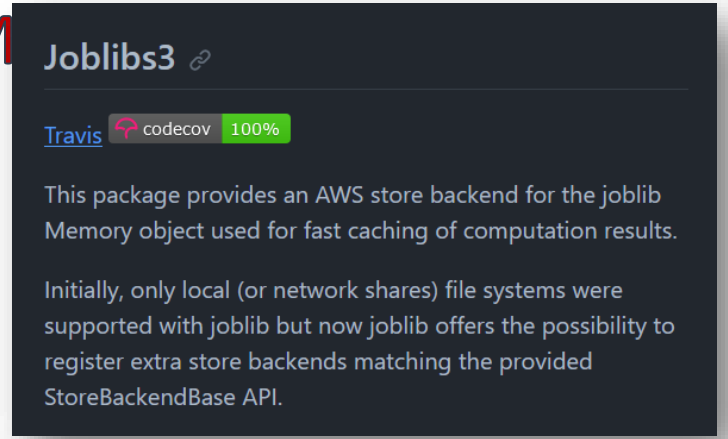
```
Memory(
    – location=None          // path/base directory
    – backend='local'        // local is regular file system
    – mmap_mode=None         // see numpy.memmap
    – compress=False         // compressed array cannot use memmap
    – verbose=1              // …
    – bytes_limit=None       // int or string like "16M"
    – backend_options=None
)
```

Memoization: joblib


```
_STORE_BACKENDS = {'local': FileSystemStoreBackend}
```

```
Mem
    location=None          // path/base directory
    backend='local'        // local is regular file system
    mmap_mode=None         // see numpy.memmap
    compress=False         // compressed array cannot use memmap
    verbose=1              // ...
    bytes_limit=None       // int or string like "16M
    backend_options=None
)
```





Memoization: joblib

```
from joblib import Mem
memory = Memory('./cach        bose=0)
```

```
@memory.cache(ignore=['foo'], verbose=2)
def memoized_function(num, foo):
    print(f"Performing even more costly calculation for {num}")
    return num * 3
```

```
memoized_function(13, foo=1)
```

```
_________________________________________________________________________________
[Memory] Calling __main__-C%3A-Users-giova-AppData-Local-Temp-ipykernel-499165028.memoized_function...
memoized_function(13, foo=1)
Performing even more costly calculation for 13
_____________________________________________memoized_function - 0.0s, 0.0min
39
```

```
memoized_function(13, foo=2)
```

```
[Memory]776.5s, 12.9min : Loading memoized_function...
39
```

Memoization: joblib

```python
def my_function(*args, **kwargs):
    print(args, kwargs)
    return True

@memory.cache(cache_validation_callback=my_function, verbose=2)
def memoized_function(num):
    print(f"Performing even more costly calculation for {num}")
    return num * 3
```

```python
memoized_function(13)
```

```
_____________________________________________________________________________
[Memory] Calling __main__-C%3A-Users-giova-AppData-Local-Temp-ipykernel-56899765.memoized_function...
memoized_function(13)
Performing even more costly calculation for 13
_____________________________________________memoized_function - 0.0s, 0.0min
39
```

```python
memoized_function(13)
```

```
({'duration': 0.001997709274291992, 'input_args': {'num': '13'}, 'time': 1697901174.566357},) {}
[Memory]41.1s, 0.7min    : Loading memoized_function...
39
```

Generic Persistence: `pickle`

```
import pickle
```

```
cache = dict()
def cached_function(num):
    if num not in cache:
        print(f"Perfoming costly calculation for {num}")
        cache[num] = num * 3 + 1
    return cache[num]
```

```
cached_function(42)
```

```
Perfoming costly calculation for 42
127
```

```
cached_function(42)
```

```
127
```

```
pickle.dump(cache, open('cache.pkl', 'wb'))
```

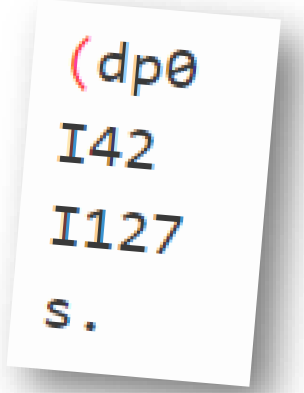
```
cache = pickle.load(open('cache.pkl', 'rb'))
```

```
cached_function(42)
```

```
127
```

Generic Persistence: `pickle`

- Supports dumping to/loading from a string (`dumps`/`loads`)
- Supports different *protocols*
  - `pickle.DEFAULT_PROTOCOL`
  - `0` (zero) for almost human-readable data
  - `pickle.HIGHEST_PROTOCOL`



Generic Persistence: `pickle`

pickle vs. joblib

```
import joblib
```

```
joblib.dump(cache, open('cache.pkl', 'wb'))
```

```
cache = joblib.load(open('cache.pkl', 'rb'))
```

```
cached_function(42)
```

```
127
```

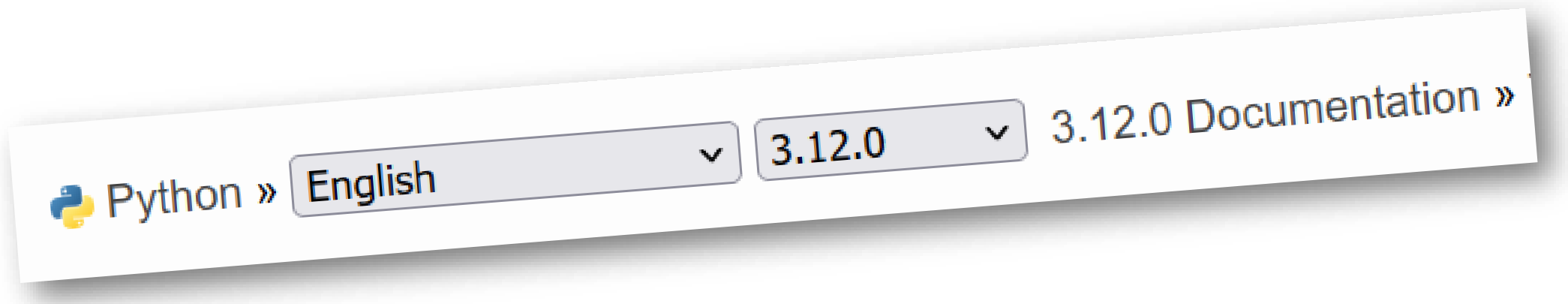
Note:

As of Python 3.8 and numpy 1.16, pickle protocol 5 introduced in PEP 574 supports efficient serialization and de-serialization for large data buffers natively using the standard library:

```
pickle.dump(large_object, fileobj, protocol=5)
```

`pickle` vs. `marshal`

- Python has a more primitive serialization module called **`marshal`**, but in general pickle should always be the preferred way to serialize Python objects
- **`marshal`** exists primarily to support Python's **`.pyc`** files



Generic Persistence: `shelve`

```
import shelve
```

```
cache = None
def shelved_function(num):
    key = str(num)
    if key not in cache:
        print(f"Perfoming costly calculation for {num}")
        cache[key] = num * 3 + 1
    return cache[key]
```

```
with shelve.open('cache.db') as cache:
    print(shelved_function(42))
    print(shelved_function(42))
```

```
Perfoming costly calculation for 42
127
127
```

```
with shelve.open('cache.db') as cache:
    print(shelved_function(42))
    print(shelved_function(42))
```

```
127
127
```

Generic Persistence: `shelve`

```
import shelve

cache = None
def shelved_functio
    key = str(num)
    if key not in cache:
        print(f"Perfoming costly calculati
        cache[key] = num * 3 + 1
    return cache[key]

with shelve.open('cache.db'
    print(shelved_function(4
    print(shelved_function(42)

Perfoming costly calculation fo
127
127

with shelve.open('cache.db') as cache:
```

The **keys** in a shelf are ordinary strings

The **values** can be anything that the pickle module can handle, including most class instances, recursive data types, and objects containing lots of shared sub-objects

cache.db.bak	14 bytes	BAK File
cache.db.dat	5 bytes	DAT File
cache.db.dir	14 bytes	DIR File

Bonus Topic: `weakref`

- A weak reference is not enough to keep an object alive
  - When the only remaining references to a referent are weak references, garbage collection may destroy the referent
- Note: Until the object is actually destroyed the weak reference may return the object

```
from dataclasses import dataclass
@dataclass
class Foo:
    num: int
```

```
x = Foo(23)
y = weakref.ref(x)
```

```
x, y, y()
```

```
(Foo(num=23), <weakref at 0x1067b9c60; to 'Foo' at 0x105f88390>, Foo(num=23))
```

Bonus Topic: weakref.proxy

```
from dataclasses import dataclass
import weakref
```

```
@dataclass
class Foo:
    num: int
```

```
store = [Foo(n) for n in range(5)]
refs = [weakref.proxy(store[n]) for n in range(5)]
```

```
refs
```

```
[<weakproxy at 0x106070590 to Foo at 0x1068eb010>,
 <weakproxy at 0x106b5cc70 to Foo at 0x106076610>,
 <weakproxy at 0x106d50220 to Foo at 0x106077190>,
 <weakproxy at 0x106d513a0 to Foo at 0x106900fd0>,
 <weakproxy at 0x106d51530 to Foo at 0x106900a10>]
```

```
del store[2]
refs
```

```
[<weakproxy at 0x106070590 to Foo at 0x1068eb010>,
 <weakproxy at 0x106b5cc70 to Foo at 0x106076610>,
 <weakproxy at 0x106d50220 to NoneType at 0x103a65d48>,
 <weakproxy at 0x106d513a0 to Foo at 0x106900fd0>,
 <weakproxy at 0x106d51530 to Foo at 0x106900a10>]
```

Bonus Topic: weakref.proxy

```
from dataclasses import dataclass
import weakref
```

```
@dataclass
class Foo:
    num: int
```

```
store = [Foo(n) for n in range(5)]
refs = [weakref.proxy(store[n]) for n in range(5)]
```

```
refs
```

```
[<weakproxy at 0x106070590 to Foo at 0x1068eb010>,
 <weakproxy at 0x106b5cc70 to Foo at 0x106076610>,
 <weakproxy at 0x106d50220 to Foo at 0x106077190>,
 <weakproxy at 0x106d513a0 to Foo at 0x106900fd0>,
 <weakproxy at 0x106d51530 to Foo at 0x106900a10>]
```

```
del store[2]
refs
```

```
[<weakproxy at 0x106070590 to Foo at 0x1068eb010>,
 <weakproxy at 0x106b5cc70 to Foo at 0x106076610>,
 <weakproxy at 0x106d50220 to NoneType at 0x103a65d48>,
 <weakproxy at 0x106d513a0 to Foo at 0x106900fd0>,
 <weakproxy at 0x106d51530 to Foo at 0x106900a10>]
```

Bonus Topic: `weak` Dictionaries

- **`WeakKeyDictionary`**
  - Mapping class that references keys weakly
- **`WeakValueDictionary`**
  - Mapping class that references values weakly
- **`keyrefs` & `valuerefs`**
- **`WeakSet`**
  - Set class that keeps weak references to its elements

More Generic Persistence

- `dbm`
  - Interfaces to the traditional NoSQL “databases” available under Unix (i.e., fast, single-keyed, key-value store)
- `sqlite3`
  - DB-API 2.0 interface for SQLite

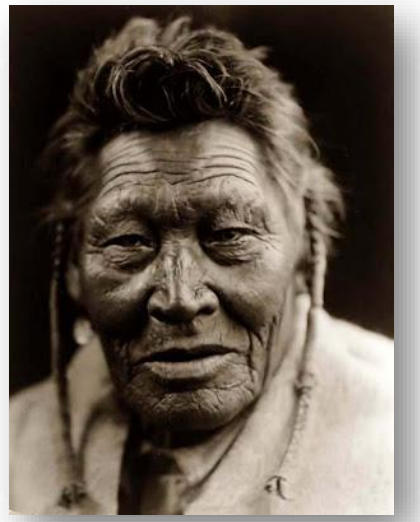
High-performance Python

Part 3: Parallel

Rule of Optimization (personal)

- Don’t write it
  - Use builtins
  - Use standard libraries
- Don’t do it
- Exploit parallel

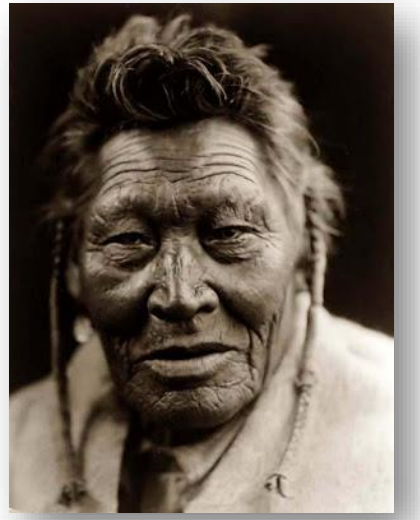
Lecture 1



Rule of Optimization (personal)

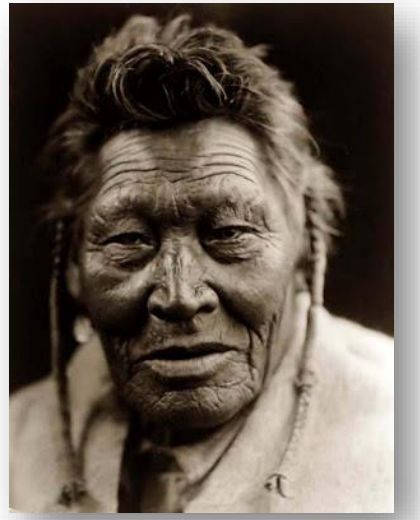
- Don’t write it
  - Use builtins
  - Use standar
- Don’t do it
  - Don’t do it yet: i.e., use lazy execution
  - Don’t do it again: i.e., cache previous calls
- Exploit parallelism

Lecture 2



Rule of Optimization (personal)

- Don’t write it
  - Use builtins
  - Use standard libraries
- Don’t do it
  - Don’t do it yet: i.e., u[illegible]on
  - Don’t do it again: i.e[illegible]evious calls
- Exploit parallelism
  - Multi thread / multi process
  - Asynchronous I/O



High-performance Python 3:
Exploiting parallelism

Basic Concepts

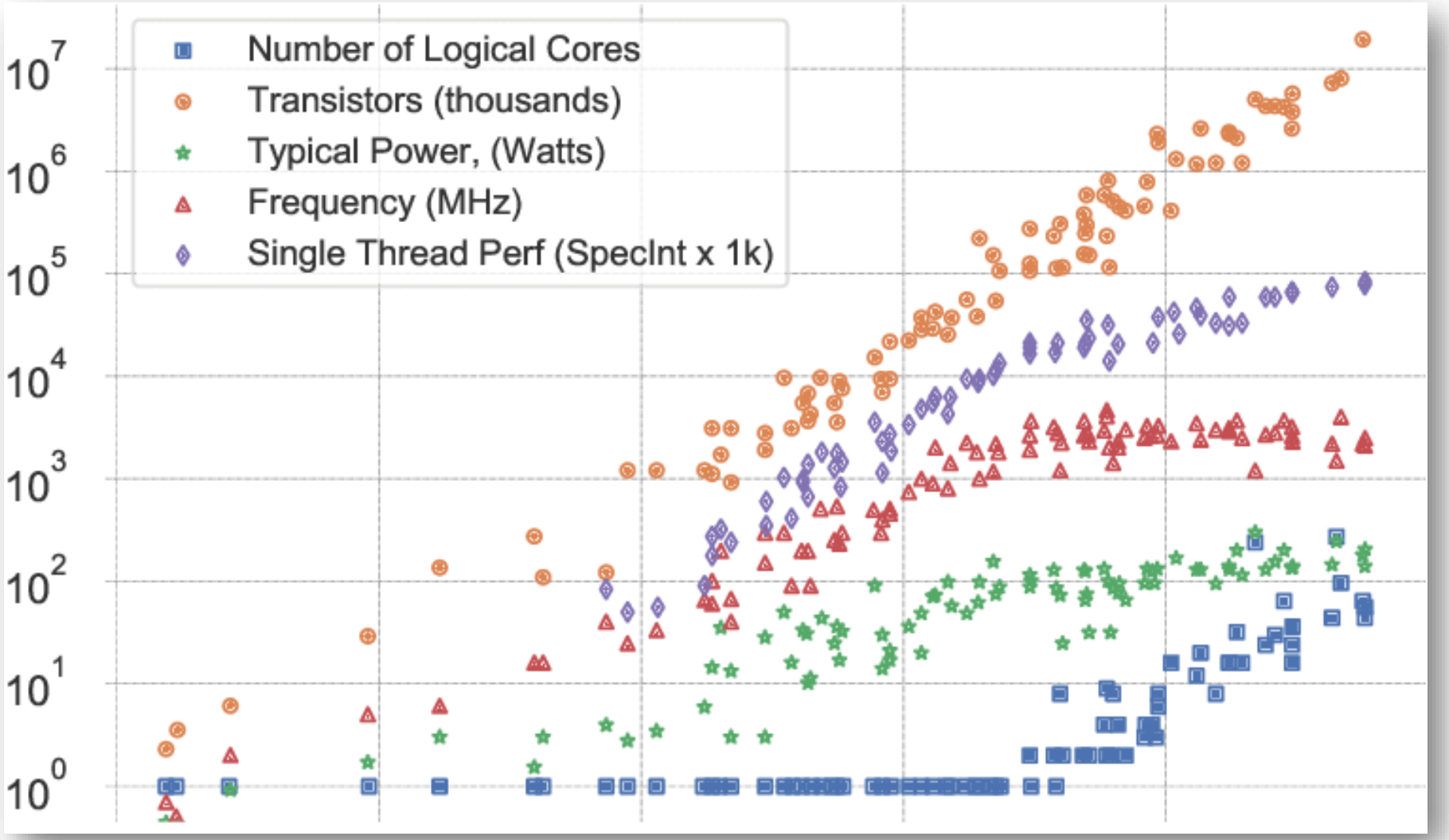
“Concurrency is not Parallelism”

- **Parallelism**
  - Simultaneous execution of (possibly related) computations
  - Two or more tasks run at the same time
  - **Doing** multiple things at the same time
- **Concurrency**
  - Composition of independently executing processes
  - Two or more tasks start and complete in overlapping time periods
  - **Dealing with** multiple things at the same time

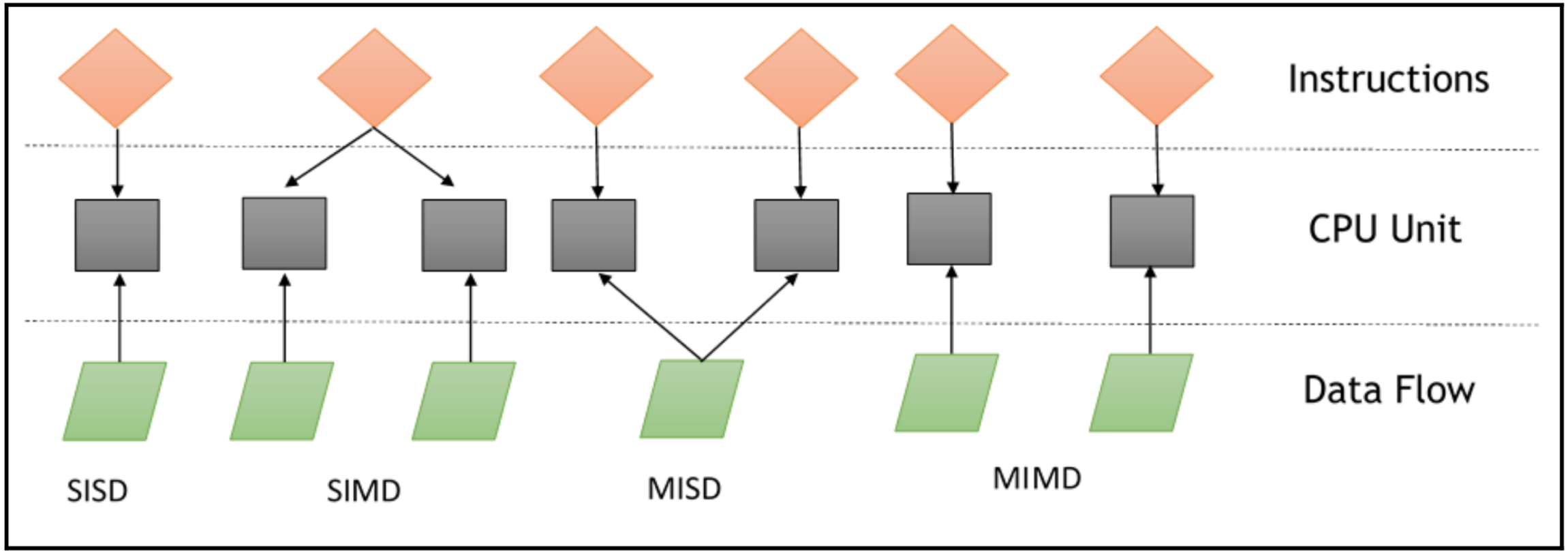
“Concurrency is not Parallelism”

- Caveats
  - The world we live in is inherently parallel
  - Concurrent algorithms do not need parallel architectures nor multiple threads
    … but a concurrent algorithm can exploit a parallel architecture!

Parallel Architectures



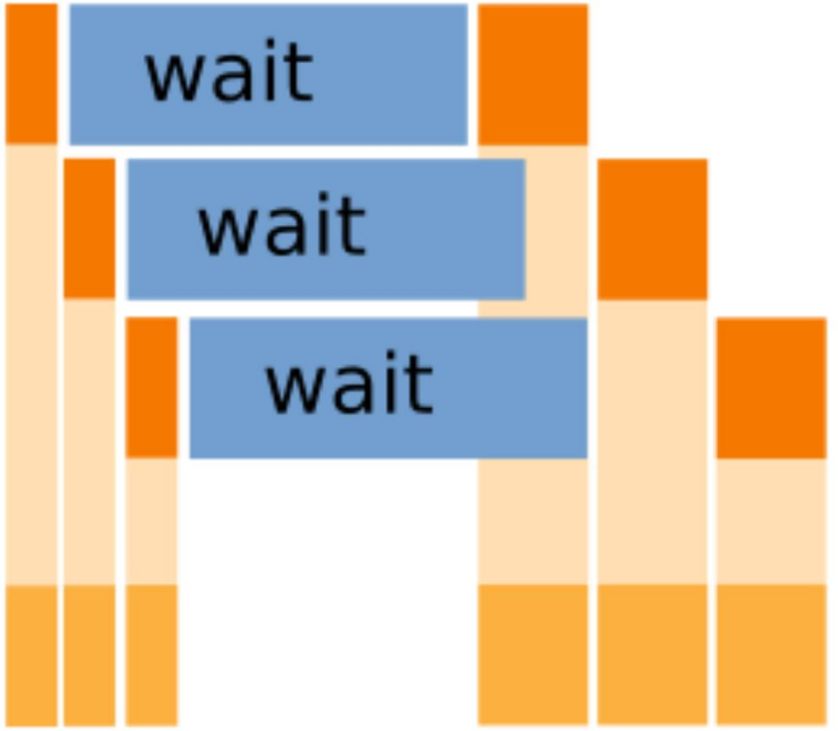
Flynn's Taxonomy



I/O bound vs. CPU bound

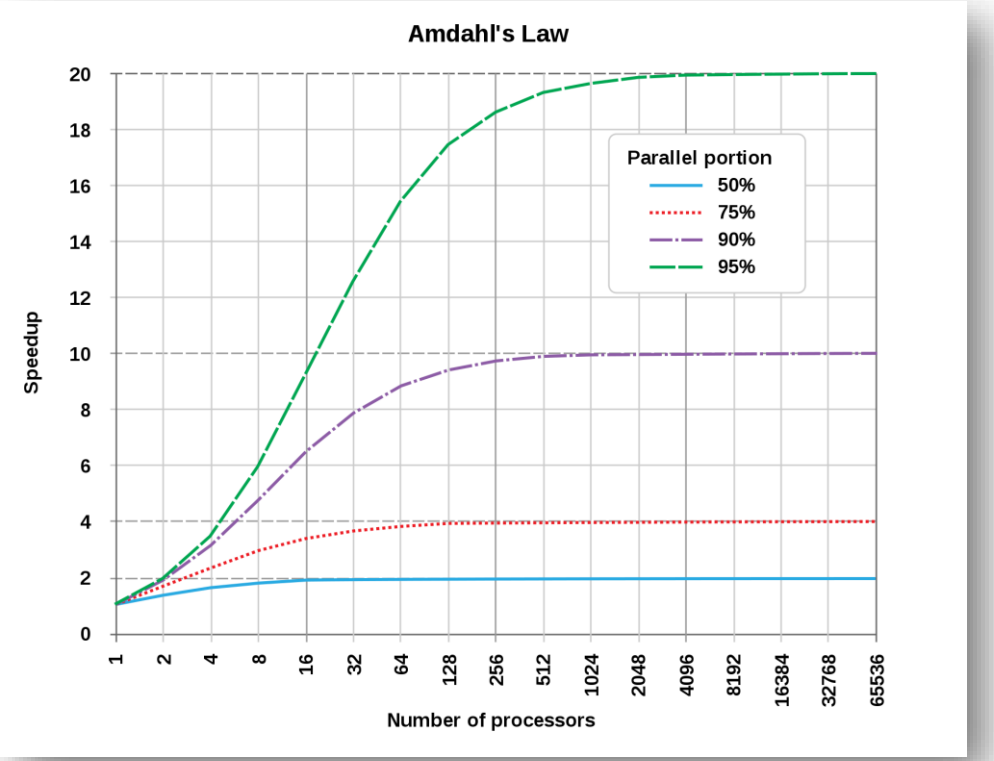
Handling ”blocking work”

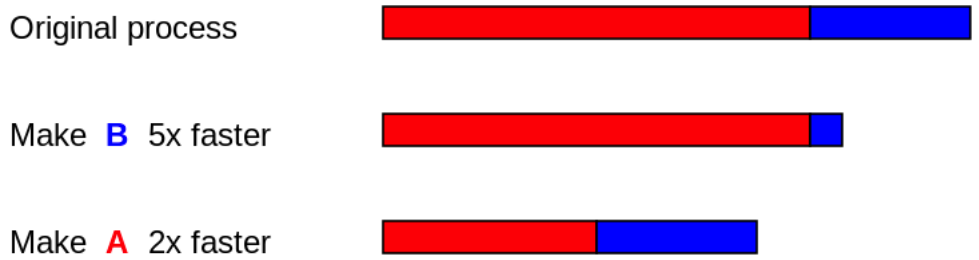
vs.



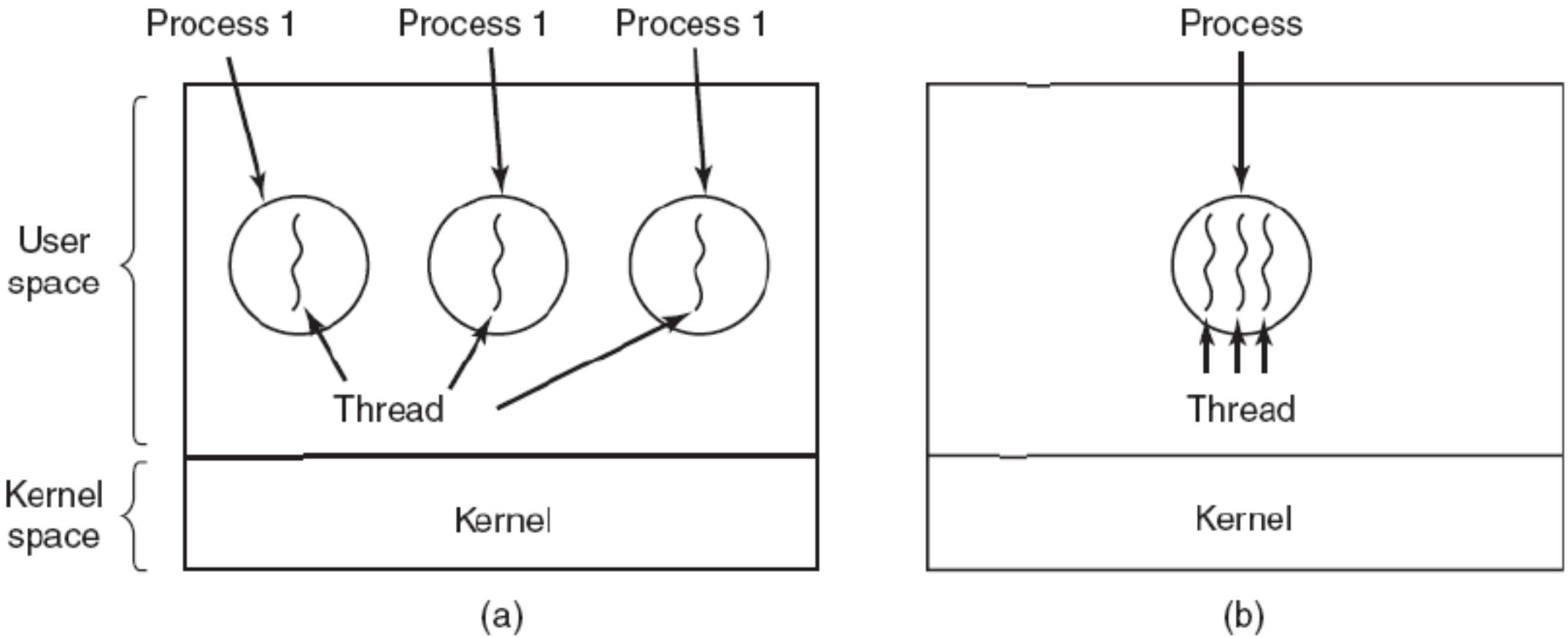
Amdahl's Law

$$T(s) = \frac{1}{(1-p) + \frac{p}{s}}$$





Thread vs. Process



Relevant Modules

- `threading` (thread-based)
  - Handles thread with a high-level interfaces built on top of the _thread module
- `multiprocessing` (process-based)
  - API similar to the threading module (both POSIX and Windows)
  - Handles both *local* and *remote* concurrency, side-steps the Global Interpreter Lock
- `subprocess` (not parallel)
  - Spawns processes, connect pipes (in/out/err), and get return codes

Relevant Modules

- `concurrent.futures` (thread- and process-based)
  - Even higher-level interface for asynchronously executing callables
- `joblib` (thread- and process-based)
  - Easy, simple parallel computing
  - Note: **not in the standard library**
- `queue` (thread-based)
  - Handles multi-producer, multi-consumer queues
  - Implements all the required locking semantics

Relevant Modules

- `asyncio` (concurrent)
  - Handles concurrent code using the async/await syntax

High-performance Python 3:
Exploiting parallelism

Thread/Process API

subprocess

```
import subprocess
```

```
def shell_task(seconds) -> int:
    return subprocess.run(["./script.sh", str(seconds)], check=True, capture_output=True)
```

```
info = shell_task(1)
info
```

```
CompletedProcess(args=['./script.sh', '1'], returncode=0, stdout=b'Process 14775 completed
after waiting 1s\n', stderr=b'')
```

```
info.stdout.decode('utf-8')
```

```
'Process 14775 completed after waiting 1s\n'
```

This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

execute_task

```
def execute_task(task_name):
    out = subprocess.run(
        f'"{SCRIPT}" "{task_name}"', check=True, capture_output=True, text=True, shell=True
    )
    logging.info(
        "execute_task: %s (thread %s/%d, pid %d) completed",
        task_name,
        threading.current_thread().name,
        threading.current_thread().native_id,
        os.getpid(),
    )
    return out.stdout.rstrip()
```

task

```
name="${1:-Gobbledygook}"

time=$((1 + RANDOM % 10))
sleep $time
echo "Task \"${name}\" completed in ${time} seconds"
```

```
IF NOT "%~1" == "" SET name=%~1
IF "%~1" == "" SET name=Gobbledygook

SET /A time=(%RANDOM%%%10)+1
TIMEOUT /T %time% /NOBREAK > NUL
ECHO Task "%name%" completed in %time% seconds
```

threading

- High-level interfaces on top of the `_thread` module
- As simple as…

```
1  thread = threading.Thread(target=execute_task, args=['Grumpy'])
2  thread.start()
3  thread.join()
```

✓ 6.0s Python

```
[14:04:34.562] INFO:execute_task: Grumpy completed
```

threading

- As simple as…

```python
threads = list()
for task in hppython3.TASKS:
    threads.append(threading.Thread(target=hppython3.execute_task, args=[task]))
for t in threads:
    t.start()
for t in threads:
    t.join()
```

✓ 7.0s

```
[01:04:49.884] INFO:execute_task: Sneezy (thread Thread-15 (execute_task)/662552, pid 41982) completed
[01:04:50.880] INFO:execute_task: Bashful (thread Thread-10 (execute_task)/662540, pid 41982) completed
[01:04:52.878] INFO:execute_task: Sleepy (thread Thread-14 (execute_task)/662550, pid 41982) completed
[01:04:53.877] INFO:execute_task: Happy (thread Thread-13 (execute_task)/662545, pid 41982) completed
[01:04:53.878] INFO:execute_task: DopeyGrumpy (thread Thread-12 (execute_task)/662544, pid 41982) completed
[01:04:55.873] INFO:execute_task: Doc (thread Thread-11 (execute_task)/662541, pid 41982) completed
```

threading

- However…

```python
def problem(data):
    logging.info("problem: Task started")
    for k in data:
        logging.info("problem: key %s -> %d", k, data[k])
        time.sleep(data[k])
    logging.info("problem: Task completed")

shared_data = {'a': 1, 'b': 2, 'c':3, 'd':4}
thread = threading.Thread(target=problem, args=[shared_data])
thread.start()
```

✓ Python

```
[18:04:38.766] INFO:problem: Task started
[18:04:38.770] INFO:problem: key a -> 1
[18:04:39.777] INFO:problem: key b -> 2
```

threading

- However…

```
1 def problem(data):
```

```
1 shared_data['j'] = 9
```

Python

```
Exception in thread Thread-6 (problem):
Traceback (most recent call last):
  File "/opt/homebrew/Cellar/python@3.12/3.12.0/Frameworks/Python.framework/Versions/3.12/lib/python3.1
    self.run()
  File "/opt/homebrew/Cellar/python@3.12/3.12.0/Frameworks/Python.framework/Versions/3.12/lib/python3.1
    self._target(*self._args, **self._kwargs)
  File "/var/folders/31/dkl97hks2c14b663vl55pt440000gn/T/ipykernel_13212/1325151708.py", line 3, in prc
RuntimeError: dictionary changed size during iteration
```

Python

```
[18:04:38.770] INFO:problem: key a -> 1
[18:04:39.777] INFO:problem: key b -> 2
```

Races

- Race Condition
  - The behavior depends on the relative timing of events (e.g., the order in which threads have been scheduled)
- Data Race
  - A specific race condition
  - More threads access some shared data concurrently, and at least one of them modifies the data

Thread-Local Data

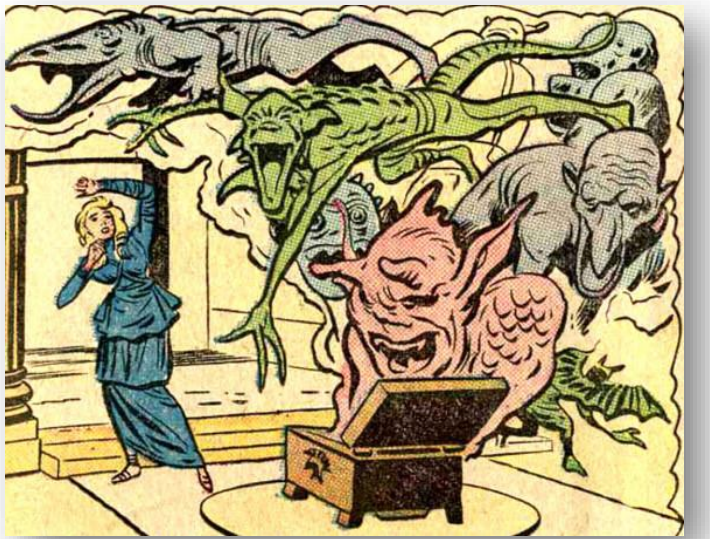
- `class threading.local`
  - Data that needs to be local to a thread

    E.g.:

```
mydata = threading.local()
mydata.number = 42
```

Locks

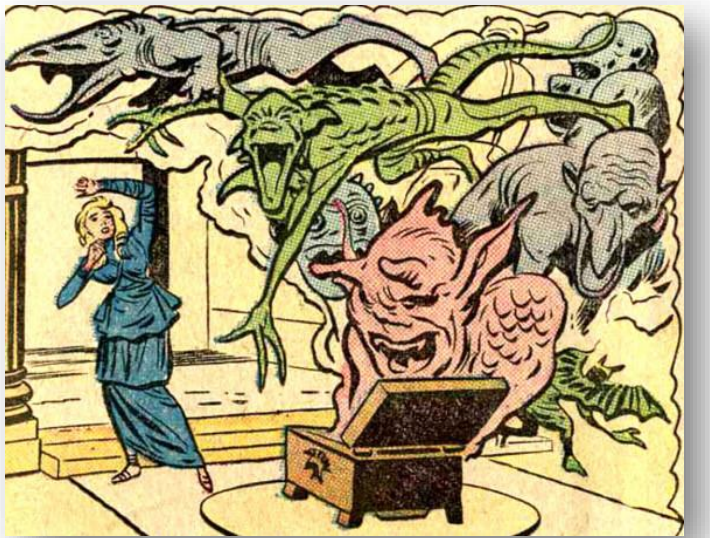
- Basic locks (`threading.Lock`)
  - `acquire` / `release`
- Reentrant locks (`threading.RLock`)
- Conditions
- Semaphores
- Events
- Timers
- Barriers



Locks

- Basic locks (`threading.Lock`)
  - `acquire` / `release`
- Reentrant locks (`threading.RLock`)
- Semaphores
- Events
- Timers
- Barriers

All of the objects provided by this module that have `acquire` and `release` methods can be used as context managers for a `with` statement. The `acquire` method will be called when the block is entered, and `release` will be called when the block is exited. Hence, the following snippet:



`multiprocessing`

- Supports spawning processes using an API similar to the `threading` module
- Offers both local and remote concurrency, effectively side-stepping the GIL
- Allows the programmer to fully leverage multiple processors on a given machine
- Runs on both POSIX and Windows

multiprocessing

```python
logging.info("*: Starting multi-process...")
processes = list()
for task in hppython3.TASKS:
    processes.append(multiprocessing.Process(target=hppython3.execute_task, args=[task]))
for p in processes:
    p.start()
for p in processes:
    p.join()
logging.info("*: multi-process completed")
```

✓ 7.1s Python

```
[01:33:23.081] INFO:*: Starting multi-process...
[01:33:24.213] INFO:execute_task: Dopey (thread MainThread/689152, pid 42730) completed
[01:33:24.223] INFO:execute_task: Grumpy (thread MainThread/689155, pid 42731) completed
[01:33:25.218] INFO:execute_task: Bashful (thread MainThread/689150, pid 42728) completed
[01:33:27.215] INFO:execute_task: Doc (thread MainThread/689151, pid 42729) completed
[01:33:27.268] INFO:execute_task: Sneezy (thread MainThread/689171, pid 42736) completed
[01:33:30.231] INFO:execute_task: Happy (thread MainThread/689158, pid 42732) completed
[01:33:30.241] INFO:execute_task: Sleepy (thread MainThread/689159, pid 42733) completed
[01:33:30.247] INFO:*: multi-process completed
```

Starting a Process

- **Spawn**
  - Available on POSIX and Windows
  - The parent process starts a fresh Python interpreter
  - The child process only inherit necessary resources. That is: **unnecessary** file descriptors and handles from the parent process **will not** be inherited.
  - Starting a process using this method is rather **slow**
  - Default on Windows and macOS

Starting a Process

- **Fork**
  - Available on POSIX
  - The parent process uses `os.fork()` to fork the Python interpreter
  - Note that safely forking a multithreaded process is **problematic**
  - Default on POSIX except macOS

Starting a Process

- **Forkserver**
  - A server process is spawned
  - Whenever a new process is needed, the parent process connects to the server and requests that it fork a new process
  - The fork server process is **single threaded** (unless system libraries or preloaded imports spawn threads as a side-effect)
  - Generally **safe**
  - No unnecessary resources are inherited

`multiprocessing` idioms

- Avoid shifting large amounts of data between processes
- Ensure that the arguments to the methods are picklable
- Do not use a proxy object from more than one thread (unless you protect it with a lock)
- Join all the processes (a.k.a., avoid Zombies)
- Better to inherit an ancestor process than pickle/unpickle
- Do not `Process.terminate` processes which use shared resources

`multiprocessing` idioms

- Make sure that the main module can be safely imported by a new Python interpreter (not really needed with *fork*)

High-performance Python 3: Exploiting parallelism

Queue

`queue`

- Multi-producer, multi-consumer queues
- Provides `Queue` (FIFO), `LifoQueue` (LIFO), and `PriorityQueue` (HPFO)
- Especially **useful in threaded programming** when information must be exchanged safely between multiple threads
- Implements **all** the required locking semantics

queue

- Check queue status (not 100% reliable!)
  `empty() / full()`
  `qsize()`
- Add an element to the queue
  `put(item, block=True, timeout=None)`
  `put_nowait(item)`
- Fetch an element from the queue
  `get(block=True, timeout=None)`
  `get_nowait()`

queue

- `task_done()`
  Indicates that a formerly enqueued task is complete (for each `get()` by a consumer thread)
- `join()`
  Blocks until all items in the queue have been gotten and processed

queue

```python
def worker(jobs):
    id = f'{threading.current_thread().name}/{threading.current_thread().native_id}'
    # queue.put(f'{id}:{task_name:02d}')
    while True:
        job = jobs.get()
        logging.info(f'worker: {id} working on {job}')
        time.sleep(job / 10)
        jobs.task_done()
```

✓ 0.0s Python

```python
running_workers = list()
for _ in range(3):
    t = threading.Thread(target=worker, args=[job_queue])
    t.start()
    running_workers.append(t)
```

queue

```
10 logging.info("*: Starting consumer/producer...")
11 for n in TASK_LIST:
12     job_queue.put(n)
13 job_queue.join()
14 logging.info("*: done")
```

✓ 18.9s Python

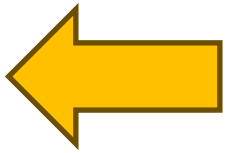
```
[08:29:10.305] INFO:*: Starting consumer/producer...
[08:29:10.307] INFO:worker: Thread-13 (worker)/813966 working on 36
[08:29:10.307] INFO:worker: Thread-14 (worker)/813967 working on 84
[08:29:10.307] INFO:worker: Thread-12 (worker)/813965 working on 0
[08:29:10.309] INFO:worker: Thread-12 (worker)/813965 working on 3
[08:29:10.611] INFO:worker: Thread-12 (worker)/813965 working on 70
[08:29:13.913] INFO:worker: Thread-13 (worker)/813966 working on 84
[08:29:17.616] INFO:worker: Thread-12 (worker)/813965 working on 40
[08:29:18.710] INFO:worker: Thread-14 (worker)/813967 working on 72
[08:29:21.622] INFO:worker: Thread-12 (worker)/813965 working on 47
[08:29:22.315] INFO:worker: Thread-13 (worker)/813966 working on 69
[08:29:29.218] INFO:*: done
```

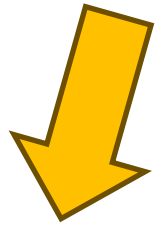
`multiprocessing.Queue`

- Shared queue implemented using a pipe and a few locks/semaphores
- Compatible with `queue.Queue`

multiprocessing.Queue

```
def parallel_queue(task_list):
    logging.info("parallel+queue: Starting multi-process...")
    processes = list()
    results = multiprocessing.Queue()
    for task in task_list:
        processes.append(
            multiprocessing.Process(target=hppython3.execute_task_queue, args=[task, results])
        )
    for p in processes:
        p.start()
    for p in processes:
        p.join()
    logging.info("parallel+queue: multi-process completed")
    print([results.get() for _ in hppython3.TASKS])
```





multiprocessing.Queue

```
[06:43:51.266] INFO:*: Starting multi-process...
[06:43:53.446] INFO:execute_task: Happy (thread MainThread/740774, pid 44141) completed
[06:43:54.442] INFO:execute_task: Sleepy (thread MainThread/740777, pid 44142) completed
[06:43:55.427] INFO:execute_task: Grumpy (thread MainThread/740771, pid 44140) completed
[06:43:57.417] INFO:execute_task: Dopey (thread MainThread/740770, pid 44139) completed
[06:44:00.409] INFO:execute_task: Doc (thread MainThread/740766, pid 44138) completed
[06:44:00.409] INFO:execute_task: Bashful (thread MainThread/740764, pid 44137) completed
[06:44:01.479] INFO:execute_task: Sneezy (thread MainThread/740781, pid 44143) completed
[06:44:01.495] INFO:*: multi-process completed
[('Happy', 'Task "Happy" completed in 2 seconds'),
 ('Sleepy', 'Task "Sleepy" completed in 3 seconds'),
 ('Grumpy', 'Task "Grumpy" completed in 4 seconds'),
 ('Dopey', 'Task "Dopey" completed in 6 seconds'),
 ('Doc', 'Task "Doc" completed in 9 seconds'),
 ('Bashful', 'Task "Bashful" completed in 9 seconds'),
 ('Sneezy', 'Task "Sneezy" completed in 10 seconds')]
```

High-performance Python 3: Exploiting parallelism

Futures

`concurrent.futures`

- High-level interface for asynchronously executing callables
- The asynchronous execution can be performed using either
  - `ThreadPoolExecutor` (threads)
  - `ProcessPoolExecutor` (as separate processes)
- Both implement the same `Executor` interface
- The asynchronous executions of a callable is encapsulated in a `Future` object

Executor

- `submit(fn, *args, **kwargs)`
  - Schedules the callable to be executed as fn(*args, **kwargs)
  - Returns a `Future` object representing the execution of the callable.
- `map(func, *iterables, timeout=None, chunksize=1)`
  - Similar to `map(func, *iterables)`
  - Not lazy

Future

- Status: `cancelled()`, `running()`, `done()`
- Return values: `result()`, `exception()`
- "Attempt to" stop the callable: `cancel()`

- Note: do not create futures directly!

High-performance Python 3: Exploiting parallelism

Joblib

joblib.Parallel

- Main helper class for readable parallel mapping, that is: parallel for loops using multiprocessing
- Generator expression converted into parallel computing

```
1  Parallel(n_jobs=-1)(delayed(execute_task)(t) for t in TASKS)
```

✓ 10.0s Python

```
[13:56:39.738] INFO:execute_task: Doc completed
[13:56:39.740] INFO:execute_task: Grumpy completed
[13:56:42.758] INFO:execute_task: Sneezy completed
[13:56:42.772] INFO:execute_task: Dopey completed
[13:56:43.755] INFO:execute_task: Bashful completed
[13:56:44.747] INFO:execute_task: Sleepy completed
[13:56:46.745] INFO:execute_task: Happy completed

['Task "Doc" completed in 3 seconds',
 'Task "Grumpy" completed in 3 seconds',
 'Task "Happy" completed in 10 seconds',
 'Task "Sleepy" completed in 8 seconds',
 'Task "Bashful" completed in 7 seconds',
 'Task "Sneezy" completed in 6 seconds',
 'Task "Dopey" completed in 6 seconds']
```

`joblib.Parallel`

- Parallelism
  - Default: **loky** backend (separate processes on separate CPUs)
  - **`backend="threading"`** (or **`prefer="threads"`**) to use threading
  - Alternative backends: **`Dask`**, **`Ray`**, **`Apache Spark`**, … ?

High-performance Python 3: Exploiting parallelism

asyncio

`asyncio`

- A library to write **concurrent** code using the **async/await** syntax (note: "*concurrent*", not "*parallel*")
- **async**
- **await**

asyncio

- High-level APIs:
  - Run coroutines concurrently with full control
  - Perform network IO and IPC
  - Control subprocesses
  - Distribute tasks via queues
  - Synchronize concurrent code

Coroutines

- I.e., “explicitly asynchronous, concurrent Pythonic code”
- More generalized form of subroutines
  - Subroutines are entered at one point and exited at another point
  - Coroutines can be entered, exited, and resumed at many different points
- Implemented with the async def statement. See also PEP 492.

asyncio

- Library to write **concurrent** code using the async/await syntax
- Perfect for **IO-bound** and high-level structured network code
- Exploited as a foundation for frameworks providing
  - High-performance network
  - Web-servers
  - Database connection libraries
  - Distributed task queues
  - …

asyncio

- Low-level APIs:
  - Create and manage event loops
  - Implement efficient protocols using transports
  - Bridge callback-based libraries and code with async/await syntax

More on async

- Awaitable: any object that can be used in an the `await`
  - E.g. a coroutine
- Asynchronous iterable (`asynch for`)
- Asynchronous context manager (`asynch with`)

Extras: starmap

```
import subprocess
```

```
def shell_task(seconds) -> int:
    return subprocess.run(["./script.sh", str(seconds)], check=True, capture_output=True)
```

```
info = shell_task(1)
info
```

```
CompletedProcess(args=['./script.sh', '1'], returncode=0, stdout=b'Process 14775 completed
after waiting 1s\n', stderr=b'')
```

```
info.stdout.decode('utf-8')
```

```
'Process 14775 completed after waiting 1s\n'
```

Extras: subprocess

```
def shell_task(seconds) -> int:
    out = subprocess.run(["./script.sh", str(seconds)], check=True, capture_output=True)
    return out.stdout.decode('utf-8')
```

```
def sequential():
    out = list()
    for rep in range(5):
        out.append(shell_task(1))
    return out
```

```
sequential()
```

```
['Process 15184 completed after waiting 1s\n',
 'Process 15186 completed after waiting 1s\n',
 'Process 15188 completed after waiting 1s\n',
 'Process 15190 completed after waiting 1s\n',
 'Process 15192 completed after waiting 1s\n']
```

```
timeit.timeit(sequential, number=1)
```

```
5.124852749999263
```

Extras: subprocess

```
def shell_task(seconds) -> int:
    out = subprocess.run(["./script.sh", str(seconds)], check=True, capture_output=True)
    return out.stdout.decode('utf-8')
```

```
def sequential():
    out = list()
    for rep in range(5):
        out.append(shell_task(1))
    return out
```

```
sequential()
```

```
['Process 15184 completed after waiting 1s\n',
 'Process 15186 completed after waiting 1s\n',
 'Process 15188 completed after waiting 1s\n',
 'Process 15190 completed after waiting 1s\n',
 'Process 15192 completed after waiting 1s\n']
```

```
timeit.timeit(sequential, number=1)
```

```
5.124852749999263
```

Low-level threading

```
def shell_task(seconds) -> int:
    out = subprocess.run(["./script.sh", str(seconds)], check=True, capture_output=True)
    msg = out.stdout.decode('utf-8')
    logging.info("shell_task(%s) on %s: completed", seconds, threading.current_thread())
    return out.stdout.decode('utf-8')
```

```
t = threading.Thread(target=shell_task, args=(1,))
t.start()
t.join()
```

```
INFO:root:shell_task(1) on <Thread(Thread-13 (shell_task), started 6263271424)>: completed
```

concurrent.futures

- Provides a high-level interface for asynchronously executing callables
- Note: Currently, there is only one module in concurrent

```
from concurrent.futures import Future, ThreadPoolExecutor
```

`Executor` and `Future`

- Two types of executors:
  - `ProcessPoolExecutor`
  - `ThreadPoolExecutor`
- The `Future` class encapsulates the asynchronous execution of a callable
  - Future instances are created by `Executor.submit()`

ThreadPoolExecutor

```
def shell_task(seconds) -> int:
    out = subprocess.run(["./script.sh", str(seconds)], check=True, capture_output=True)
    msg = out.stdout.decode('utf-8')
    logging.info("shell_task(%s) on %s: completed", seconds, threading.current_thread())
    return out.stdout.decode('utf-8')
```

```
def thread_pool():
    with ThreadPoolExecutor(max_workers=4) as executor:
        for result in executor.map(shell_task, [1, 1, 1]):
            logging.info("MAIN: %s", result[:-1])
```

```
timeit.timeit(thread_pool, number=1)
```

```
INFO:root:shell_task(1) on <Thread(ThreadPoolExecutor-3_1, started 6280097792)>: completed
INFO:root:shell_task(1) on <Thread(ThreadPoolExecutor-3_0, started 6263271424)>: completed
INFO:root:shell_task(1) on <Thread(ThreadPoolExecutor-3_2, started 6296924160)>: completed
INFO:root:MAIN: Process 15675 completed after waiting 1s
INFO:root:MAIN: Process 15674 completed after waiting 1s
INFO:root:MAIN: Process 15676 completed after waiting 1s
```

```
1.0310391659986635
```

ProcessPoolExecutor

giovanni.squillero@polito.it