Performance Optimization Techniques

Cuprins

Ce este profilarea și când să o folosești
Instrumente pentru măsurarea performanței în Python

Cuprins

Analiza complexității timpului de execuție
Tehnici de optimizare

Cuprins

Structuri de date
Evitarea utilizării ineficiente a structurilor de date

Cuprins

Explicarea conceptelor de paralelism și concurență
Utilizarea thread-urilor și proceselor în Python
MapReduce example

Profilarea și Măsurarea Performanței – Ce este profilarea și când să o folosești

Profilarea este un proces de analiză și măsurare a performanței unui program sau a unei aplicații pentru a identifica zonele de cod care consumă resursele (cum ar fi timpul de execuție sau memoria) și pentru a le optimiza.

Profilarea și Măsurarea Performanței – Ce este profilarea și când să o folosești

Scopul profilării este de a identifica și corecta bottleneck-urile și ineficiențele din codul tău pentru a face aplicația să ruleze mai rapid și să folosească mai eficient resursele disponibile.

Există mai multe tool-uri și module Python pentru profilare, fiecare cu caracteristici specifice.

- cProfile
- line_profiler
- memory_profiler
- **SnakeViz**

Alegerea unei unelte de profilare depinde de necesitățile noastre specifice și de tipul de analiză pe care dorim să le efectuăm (timp de execuție, utilizare a memoriei, profilare de linie etc.)

tottime este timpul total petrecut numai în funcție (total time)

cumtime este timpul total petrecut în funcție plus toate funcțiile pe care le-a numit această funcție. (cumulative time)

ncalls este numărul total de apeluri către funcție.

cProfile

```
import cProfile

from utils import get_random_list

if __name__ == "__main__":
    cProfile.run("get_random_list()")
```

```
         72605440 function calls in 8.896 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002    8.896    8.896 <string>:1(<module>)
 10000000    2.320    0.000    3.063    0.000 random.py:235(_randbelow_with_getrandbits)
 10000000    2.890    0.000    6.552    0.000 random.py:367(choice)
        1    0.041    0.041    8.894    8.894 utils.py:18(get_random_list)
 10100000    1.739    0.000    8.292    0.000 utils.py:21(<genexpr>)
        1    0.000    0.000    8.896    8.896 {built-in method builtins.exec}
 20000000    0.599    0.000    0.599    0.000 {built-in method builtins.len}
   100000    0.006    0.000    0.006    0.000 {method 'append' of 'list' objects}
 10000000    0.297    0.000    0.297    0.000 {method 'bit_length' of 'int' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
 12305436    0.446    0.000    0.446    0.000 {method 'getrandbits' of '_random.Random' objects}
   100000    0.555    0.000    8.847    0.000 {method 'join' of 'str' objects}
```

line_profiler

```
import random
import string
from line_profiler import LineProfiler

lp = LineProfiler()

def test_function(num_pairs=100000, max_length=100):
    random_list = []
    for _ in range(num_pairs):
        random_list.append(''.join(random.choice(string.ascii_letters) for _ in range(max_length)))
    return random_list

if __name__ == "__main__":
    lp.add_function(test_function)
    lp.run("test_function()")
    lp.print_stats()
```

```
Timer unit: 1e-09 s

Total time: 5.76019 s
File: /home/andrei/Desktop/masterclass/02_01_01_line_profiler.py
Function: test_function at line 8

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           def test_function(num_pairs=100000, max_length=100):
     9         1        446.0    446.0      0.0      random_list = []
    10    100001    7985867.0     79.9      0.1      for _ in range(num_pairs):
    11    100000 5752205026.0  57522.1     99.9          random_list.append(''.join(random.choice(string.ascii_letters) for _ in range(max_length)))
    12         1         99.0     99.0      0.0      return random_list
```

memory_profiler

```
import random
import string
from memory_profiler import profile

@profile
def test_function(num_pairs=10000, max_length=100):
    random_list = []
    for _ in range(num_pairs):
        random_list.append(''.join(random.choice(string.ascii_letters) for _ in range(max_length)))
    return random_list

if __name__ == "__main__":
    test_function()
```

```
Filename: /home/andrei/Desktop/masterclass/02_01_01_memory_profiler.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     5     21.4 MiB     21.4 MiB           1   @profile
     6                                         def test_function(num_pairs=10000, max_length=100):
     7     21.4 MiB      0.0 MiB           1       random_list = []
     8     23.0 MiB      0.0 MiB       10001       for _ in range(num_pairs):
     9     23.0 MiB      1.6 MiB     2030000           random_list.append(''.join(random.choice(string.ascii_letters) for _ in range(max_length)))
    10     23.0 MiB      0.0 MiB           1       return random_list
```

SnakeViz

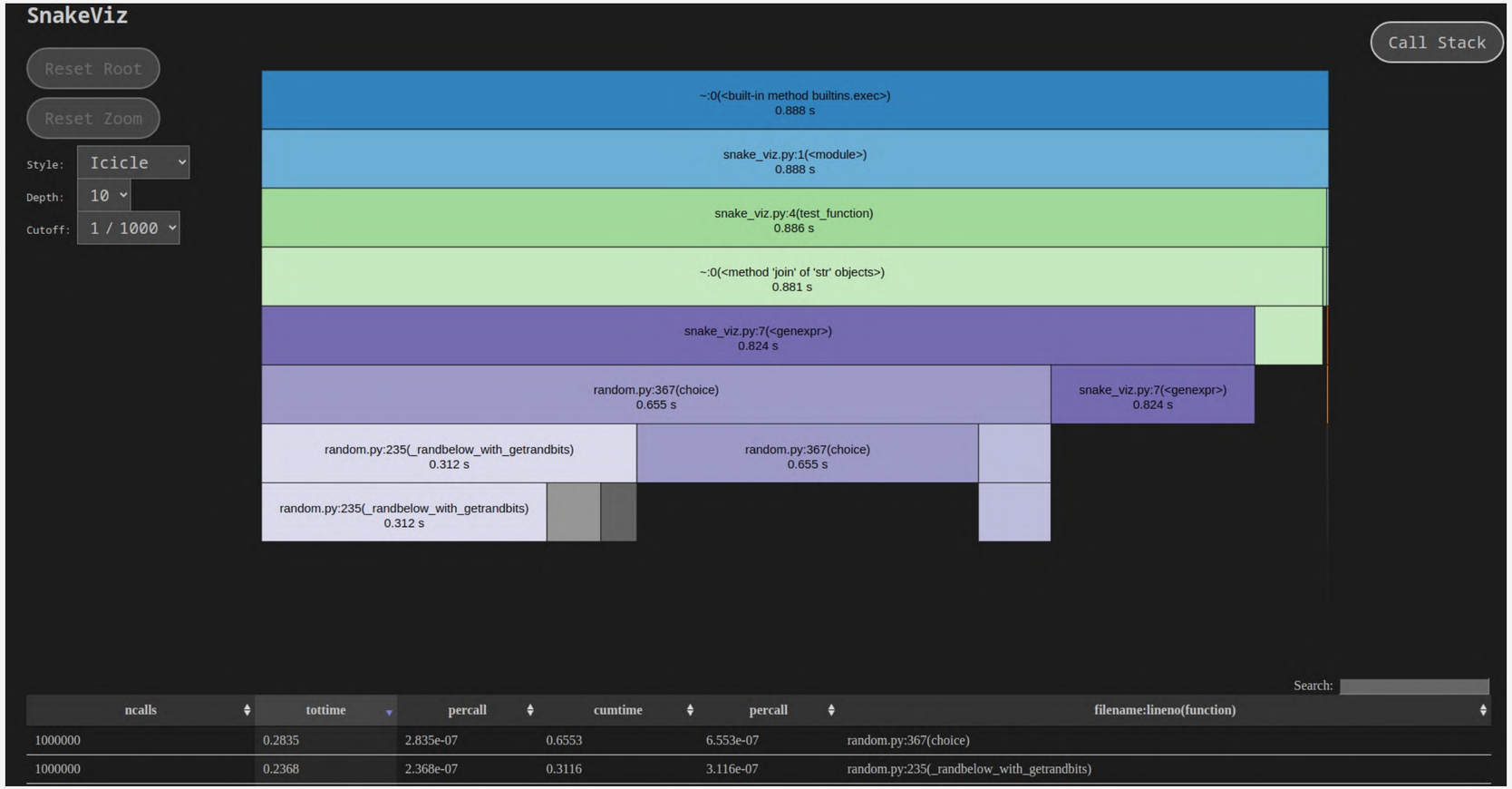
```
import random
import string

def test_function(num_pairs=10000, max_length=100):
    random_list = []
    for _ in range(num_pairs):
        random_list.append(''.join(random.choice(string.ascii_letters) for _ in range(max_length)))
    return random_list

if __name__ == "__main__":
    test_function()
```

```
python -m cProfile -o 02_01_01_snakeviz.prof 02_01_01_snakeviz.py
snakeviz 02_01_01_snakeviz.prof
```

SnakeViz



Optmizare algoritmică - Analiza complexității timpului de execuție

Una dintre cele mai eficiente modalități de îmbunătățire a performanței aplicațiilor este prin utilizarea unor algoritmi și structuri de date mai bune.

Optmizare algoritmică - Analiza complexității timpului de execuție

Îmbunătățirile aduse algoritmilor sunt extrem de eficiente pentru îmbunătățirea performanței, deoarece, în general, permit aplicației să gestioneze mai eficient seturi de date din ce în ce mai mari.

Optmizare algoritmică - Analiza complexității timpului de execuție

Notația "O" descrie modul în care timpul de rulare sau necesarul de memorie a unui algoritm crește în ceea ce privește dimensiunea intrării. Din acest motiv, o complexitate mai mică denotă un algoritm mai eficient.

Optmizare algoritmică - Analiza complexității timpului de execuție

O complexitate "O(1)" semnifică faptul că algoritmului este de complexitatea constantă, adică timpul de execuție nu depinde de mărimea datelor de intrare.

Optmizare algoritmică - Analiza complexității timpului de execuție

De exemplu, accesul la un element dintr-un array sau o listă în Python are complexitatea "O(1)", deoarece nu contează câte elemente sunt în array, timpul necesar pentru a accesa un element este fix și nu crește odată cu mărimea array-ului.

Optmizare algoritmică – Analiza complexității timpului de execuție

```
input = list(range(10))
for i, _ in enumerate(input):
    input[i] += 1
```

Optmizare algoritmică – Analiza complexității timpului de execuție

O înțelegere de bază a complexității structurilor de date din Python este crucială pentru scrierea unui cod eficient.

for vs for&yield vs map

```
def without_map():
    _our_list = []
    for i in original_string:
        _our_list.append(i.upper())
    return _our_list

def without_map_and_yield():
    for i in original_string:
        yield i.upper()

def with_map():
    return map(str.upper, our_string)
```

```
0.00000411598011851311 seconds - without_map
0.00000029098009690642 seconds - without_map_and_yield
0.00000058696605265141 seconds - with_map
```

short string

```
0.00005484797293320298 seconds - without_map
0.00000045000342652202 seconds - without_map_and_yield
0.00000072101829573512 seconds - with_map
```

longer string

deque

```
def process_data(_data_without_map):
    for i in data_without_map:
        pass
```

```
0.00002858799416571856 seconds - process_data
0.00001067802077159286 seconds - process_data
0.00001015397720038891 seconds - process_data
```

without deque

```
def process_data_deque(_data_without_map):
    for i in deque(data_without_map):
        pass
```

```
0.00001006899401545525 seconds - process_data_deque
0.00000998604809865355 seconds - process_data_deque
0.00000989600084722042 seconds - process_data_deque
```

with deque

for try except

```
def check_key_in_dict(_alphabet_dict):
    for w in masterclass_word:
        if w not in _alphabet_dict:
            _alphabet_dict[w] = 0
        _alphabet_dict[w] += 1
    return _alphabet_dict

def check_key_in_dict_with_try(_alphabet_dict):
    for w in _alphabet_dict:
        try:
            _alphabet_dict[w] += 1
        except KeyError:
            _alphabet_dict[w] = 1
    return alphabet_dict
```

```
0.00001082895323634148 seconds - check_key_in_dict
0.00000232702586799860 seconds - check_key_in_dict_with_try
```

for try except

```
def check_key_in_dict(_alphabet_dict):
    for w in masterclass_word:
        if w not in _alphabet_dict:
            _alphabet_dict[w] = 0
        _alphabet_dict[w] += 1
    return _alphabet_dict

def check_key_in_dict_with_try(_alphabet_dict):
    for w in _alphabet_dict:
        try:
            _alphabet_dict[w] += 1
        except KeyError:
            _alphabet_dict[w] = 1
    return alphabet_dict
```

```
0.00001082895323634148 seconds - check_key_in_dict
0.00000232702586799860 seconds - check_key_in_dict_with_try
```

Notă!
putem face și mai optim dar acest lucru îl vom vedea în câteva minute.

local variable

```
class Masterclass:
    def optimization(self, x):
        # print(x*x)
        pass

def call_class_method():
    for i in range(100):
        Masterclass().optimization(2)

def call_class_method_with_variable():
    MasterclassObj = Masterclass()
    optimization = MasterclassObj.optimization
    for i in range(100):
        optimization(2)
```

```
0.00000101403566077352 seconds - call_class_method
0.00000076601281762123 seconds - call_class_method_with_variable
```

single call

```
0.00000862701563164592 seconds - call_class_method
0.00000360398553311825 seconds - call_class_method_with_variable
```

loop call

Optmizare algoritmică - Tehnici de optimizare

more ...

concatenează folosind cu join

Optmizare algoritmică - Tehnici de optimizare

more ...

concatenează folosind cu join

generator expressions în loc de list comprehensions

more ...

concatenează folosind cu join

generator expressions în loc de list comprehensions

NumPy arrays în loc de liste

Optmizare algoritmică - Tehnici de optimizare

more ...

concatenează folosind cu join

generator expressions în loc de list comprehensions

NumPy arrays în loc de liste

evită folosirea 'for in for'

Optmizare algoritmică - Tehnici de optimizare

more ...

concatenează folosind cu join

generator expressions în loc de list comprehensions

NumPy arrays în loc de liste

evită folosirea 'for in for'

folosește operatorii built-in

Optmizare algoritmică - Tehnici de optimizare

more ...

concatenează folosind cu join

generator expressions în loc de list comprehensions

NumPy arrays în loc de liste

evită folosirea 'for in for'

folosește operatorii built-in

lazy imports

Optmizare algoritmică - Tehnici de optimizare

more ...

concatenează folosind cu join

generator expressions în loc de list comprehensions

NumPy arrays în loc de liste

evită folosirea 'for in for'

folosește operatorii built-in

lazy imports

utilizează structuri de date adecvate

Utilizarea Corectă a Structurilor de Date – Utilizarea Corectă a Structurilor de Date

Lists and deques

Dictionaries

Sets

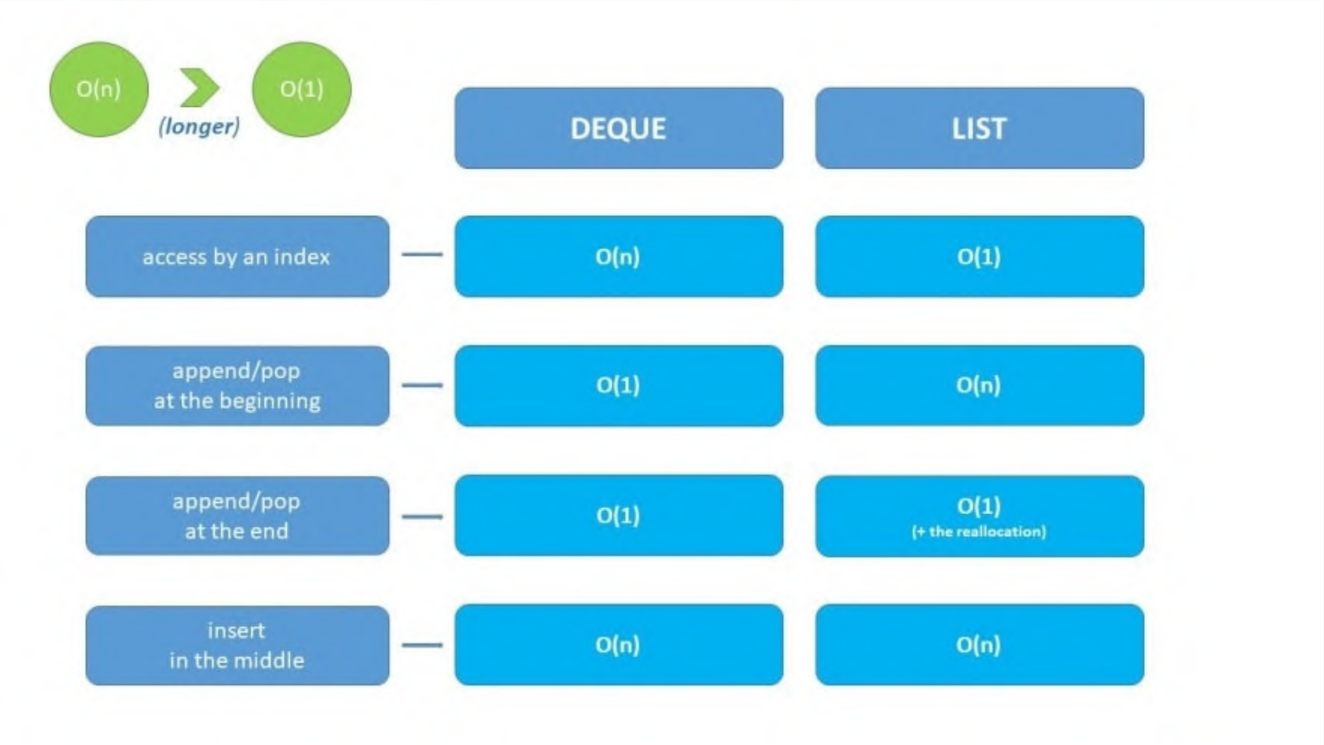
Heaps

Tries

Lists

Lists and deques

Lists and deques



https://dev.to/v_it_aly/python-deque-vs-listwh-25i9
https://dev.to/wnleao/python-deque-vs-list-time-comparison-5ch4

Lists and deques

Code	N=10000 (µs)	N=20000 (µs)	N=30000 (µs)	Time
`list.pop()`	0.50	0.59	0.58	$O(1)$
`list.pop(0)`	4.20	8.36	12.09	$O(N)$
`list.append(1)`	0.43	0.45	0.46	$O(1)$
`list.insert(0, 1)`	6.20	11.97	17.41	$O(N)$

The speed of different list operations

Code	N=10000 (µs)	N=20000 (µs)	N=30000 (µs)	Time
`deque.pop()`	0.41	0.47	0.51	$O(1)$
`deque.popleft()`	0.39	0.51	0.47	$O(1)$
`deque.append(1)`	0.42	0.48	0.50	$O(1)$
`deque.appendleft(1)`	0.38	0.47	0.51	$O(1)$

The speed of different deque operations

Code	N=10000 (µs)	N=20000 (µs)	N=30000 (µs)	Time
`deque[0]`	0.37	0.41	0.45	O(1)
`deque[N - 1]`	0.37	0.42	0.43	O(1)
`deque[int(N / 2)]`	1.14	1.71	2.48	O(N)

The inefficiency of deques in accessing the middle element

Lists and deques - Indexul unui element

Căutarea indexului unui element într-o listă este în general o operațiune O(N) și se realizează folosind metoda **list.index**.

O modalitate simplă de a accelera căutările în liste este să menții lista sortată și să efectuezi o căutare binară folosind modulul **bisect**.

Lists and deques – Indexul unui element

```
@timeit
def list_search(_random_list):
    return _random_list.index('aa')
```

Lists and deques – Indexul unui element

```
@timeit
def list_search(_random_list):
    return _random_list.index('aa')
```

```
@timeit
def bisect_search(_random_list):
    return bisect.bisect(_random_list, 'aa')
```

Lists and deques – Indexul unui element

```
@timeit
def list_search(_random_list):
    return _random_list.index('aa')
```

```
@timeit
def bisect_search(_random_list):
    return bisect.bisect(_random_list, 'aa')
```

```
sorted list
0.000009759038221091 03 seconds - list_search
0.000003693974576890 47 seconds - deques_search
```

```
unsorted list
0.000037941965274512 77 seconds - list_search
0.000003942986950278 28 seconds - deques_search
```

Code	N=10000 (µs)	N=20000 (µs)	N=30000 (µs)	Time
list.index(a)	87.55	171.06	263.17	*O*(N)
index_bisect(list, a)	3.16	3.20	4.71	*O*(log(N))

The efficiency of the bisect function

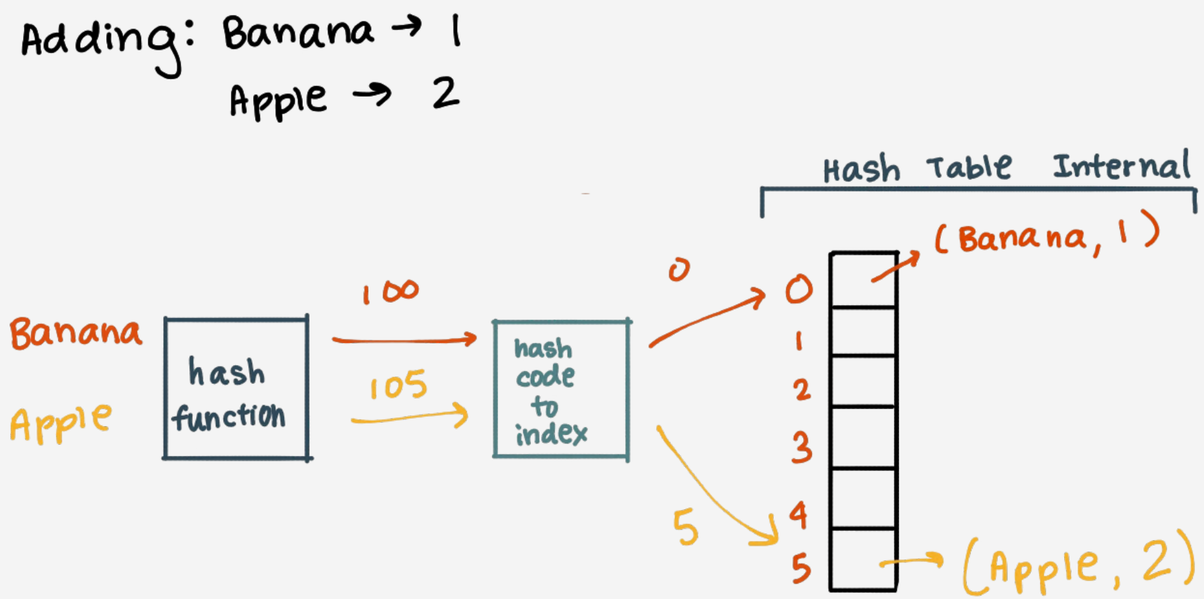
https://docs.python.org/3/library/bisect.html

Dictionaries & Hash Map

Dicționarele sunt implementate ca hash map și sunt foarte bune la inserarea, ștergerea și accesul elementelor.

Toate aceste operații au o complexitate medie în timp O(1).

Dictionaries & Hash Map



Dictionaries & Hash Map

```
class CustomDict:
    def __init__(self):
        self.size = 10
        self.table = [None] * self.size
    def hash_function(self, key):
        return hash(key) % self.size
    def set(self, key, value):
        index = self.hash_function(key)
        self.table[index] = (key, value)
```

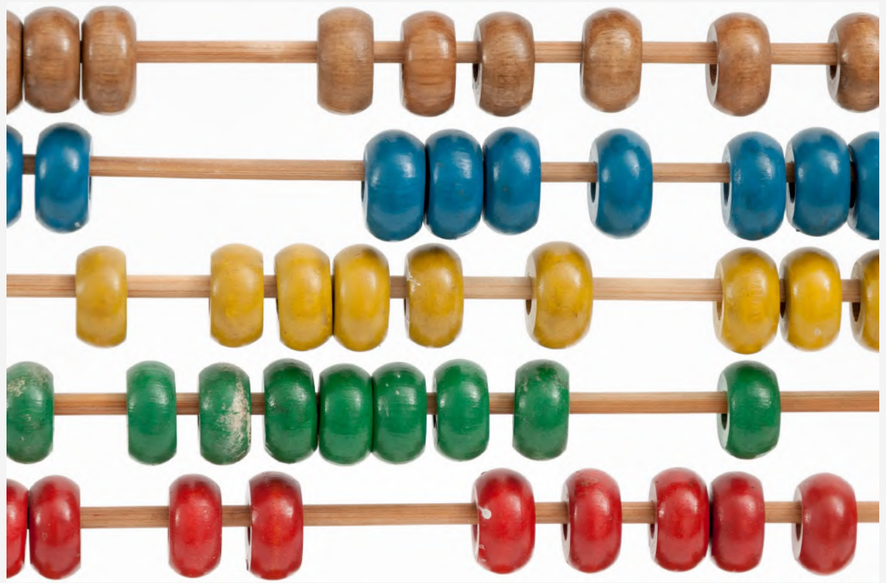
```
0.00000421400181949139 seconds - call_custom_dict
0.00000089500099420547 seconds - call_default_dict
```

```
def get(self, key):
    index = self.hash_function(key)
    entry = self.table[index]
    if entry is not None and entry[0] == key:
        return entry[1]
    return None

def remove(self, key):
    index = self.hash_function(key)
    entry = self.table[index]
    if entry is not None and entry[0] == key:
        self.table[index] = None
    raise KeyError(f"Key '{key}' not found")
```

Dictionaries & Counter

Un dicționar poate fi folosit pentru a număra eficient elementele unice dintr-o listă.



Dictionaries & Counter

```
def counter_dict(_random_list):
    counter = {}
    for item in _random_list:
        if item not in counter:
            counter[item] = 1
        else:
            counter[item] += 1
    return counter
```

Dictionaries & Counter

```
def counter_dict(_random_list):
    counter = {}
    for item in _random_list:
        if item not in counter:
            counter[item] = 1
        else:
            counter[item] += 1
    return counter
```

```
def counter_defaultdict(_random_list):
    counter = defaultdict(int)
    for item in _random_list:
        counter[item] += 1
    return counter
```

```
0.009512530057691037 65 seconds - counter_dict
0.009854383068159222 60 seconds - counter_defaultdict
```

Dictionaries & Counter

```
def counter_dict(_random_list):
    counter = {}
    for item in _random_list:
        if item not in counter:
            counter[item] = 1
        else:
            counter[item] += 1
    return counter
```

```
def counter_defaultdict(_random_list):
    counter = defaultdict(int)
    for item in _random_list:
        counter[item] += 1
    return counter
```

```
def counter_default(_random_list):
    from collections import Counter
    return Counter(_random_list)
```

```
0.009512530057691037 65 seconds - counter_dict
0.009854383068159222 60 seconds - counter_defaultdict
0.004213898908346891 40 seconds - counter_default
```

Dictionaries & Counter vs Set

```
def counter_dict(_random_list):
    counter = {}
    for item in _random_list:
        if item not in counter:
            counter[item] = 1
        else:
            counter[item] += 1
    return counter
```

```
def counter_defaultdict(_random_list):
    counter = defaultdict(int)
    for item in _random_list:
        counter[item] += 1
    return counter
```

```
def counter_default(_random_list):
    from collections import Counter
    return Counter(_random_list)
```

```
0.009512530057691037 65 seconds - counter_dict
0.009854383068159222 60 seconds - counter_defaultdict
0.004213898908346891 40 seconds - counter_default
0.002974483068101108 07 seconds - counter_with_set
```

```
def counter_with_set(_random_list):
    return len(set(_random_list))
```

Dictionaries & Counter vs Set

len(set(_random_list)) din exemplul dat efectuează doar o operație de conversie a listei într-un set și numărarea elementelor unice

Counter numără apariția tuturor elementelor din listă, ceea ce poate necesita mai mult timp în funcție de dimensiunea listei

Dictionaries & Inverted index

```
str_list = ["astazi suntem la masterclass",
            "avem topic-uri interesante la masterclass",
            "python masterclass este ceva wow",
            "la masterclass am vazut lucruri noi",
            "da", ...]
```

Dictionaries & Inverted index

```
str_list = ["astazi suntem la masterclass",
            "avem topic-uri interesante la masterclass",
            "python masterclass este ceva wow",
            "la masterclass am vazut lucruri noi",
            "da", ...]
matches = [s for s in str_list if "masterclass" in s]
```

Dictionaries & Inverted index

```
data = ["astazi suntem la masterclass",
        "avem topic-uri interesante la masterclass",
        "python masterclass este ceva wow",
        "la masterclass am vazut lucruri noi",
        "da", ...]
matches = [s for s in data if "masterclass" in s]
```

Dictionaries & Inverted index

```
for i, doc in enumerate(self._data):
    for word in doc.split():
        try:
            self.index[word].append(i)
        except KeyError:
            self.index[word] = [i]
```

Dictionaries & Inverted index

```
for i, doc in enumerate(self._data):
    for word in doc.split():
        self.index.setdefault(word, []).append(i)
```

Dictionaries & Inverted index

```
for i, doc in enumerate(self._data):
    for word in doc.split():
        self.index.setdefault(word, []).append(i)
```

```
results = self.index[word]
matches = [self._data[i] for i in results]
```

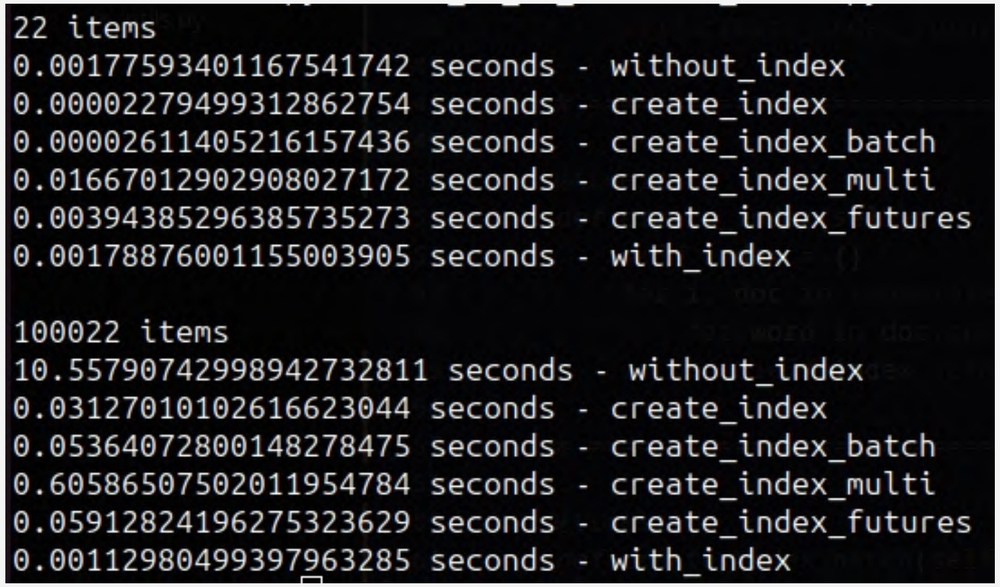
```
22 items
0.0018938019638881 0873 seconds - without_index
0.0007310259970836 3414 seconds - with_index

100022 items
10.609200889011844 99264 seconds - without_index
0.0006368529866449 5349 seconds - with_index
```

Dictionaries & Inverted index

Batch Insertion

Parallelism



Rezultatele pot fi mult influențate de tipul de date, de volum și ordinea în care sunt.

Sets

În Python, seturile sunt implementate folosind un algoritm bazat pe hash, la fel ca și dicționarele

prin urmare, complexitățile de timp pentru adăugare, ștergere și testare pentru apartenență la scară ca O(1) cu dimensiunea colecției.

Code	Time
`s.union(t)`	$O(S + T)$
`s.intersection(t)`	$O(min(S, T))$
`s.difference(t)`	$O(S)$

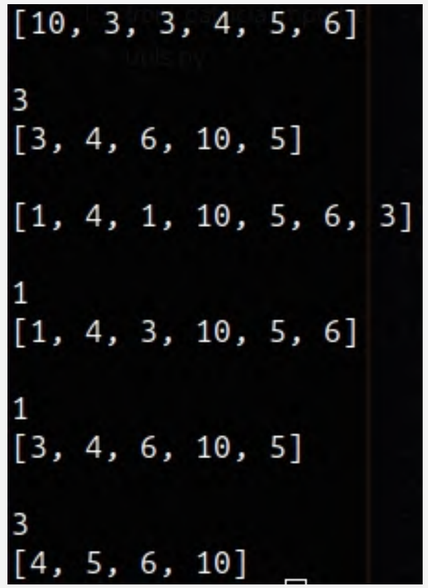
The running time of set operations

Heaps

Heap-urile sunt structuri de date concepute pentru a găsi și extrage rapid valori minime sau maxime dintr-o colecție.

Heaps

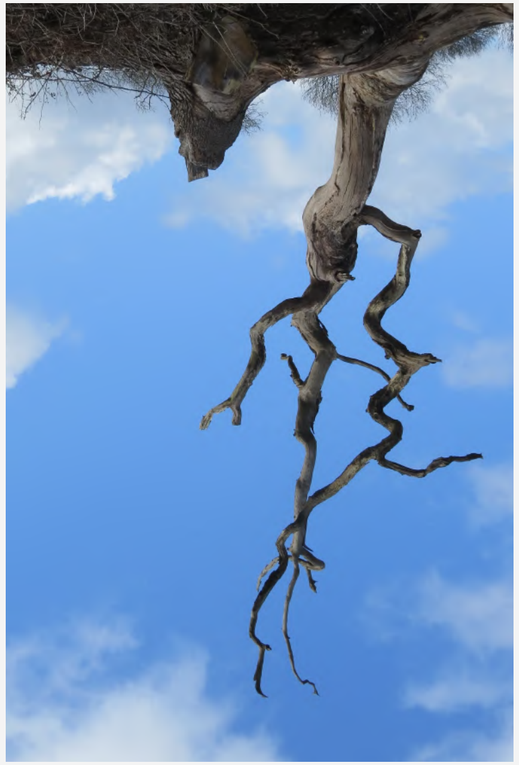
```
import heapq
collection = [10, 3, 3, 4, 5, 6]
heapq.heapify(collection)
result = heapq.heappop(collection)
heapq.heappush(collection, 1)
heapq.heappush(collection, 1)
result = heapq.heappop(collection)
result = heapq.heappop(collection)
result = heapq.heappop(collection)
```



https://www.geeksforgeeks.org/max-heap-in-python/

Tries

Arborii sunt extrem de rapizi la potrivirea unei liste de șiruri cu un prefix.



Python nu include o implementare pentru arbori, dar putem apela la libraria **marisa-trie**.

Tries

```
@timeit
def default_tries_matches(_strings, search_key='AA'):
    return [s for s in _strings if s.startswith(search_key)]
```

Tries

```
@timeit
def default_tries_matches(_strings, search_key='AA'):
    return [s for s in _strings if s.startswith(search_key)]


class SearchTrie():
    def __init__(self, strings):
        self.trie = marisa_trie.Trie(strings)
    def init_strings_dict(self, strings):
        for string in strings:
            self.trie[string] = string
    @timeit
    def search(self, prefix='AA'):
        return self.trie.prefixes(prefix)
```

```
0.00297065201448276639 seconds - default_tries_matches
0.00000360101694241166 seconds - search
```

Utilizarea Corectă a Structurilor de Date – Evitarea utilizării ineficiente a structurilor de date

identificați procesele care vor fi cele mai frecvente: ștergere, adăugare, căutare etc.

identificați cum vor fi salvate datele sau în ce mod sunt necesare: sortate, nesortate etc.

identificați cât de rapid aveți nevoie de ele.

Utilizarea Corectă a Structurilor de Date – Evitarea utilizării ineficiente a structurilor de date

evitați calculele repetitive și dacă se poate folosiți preprocesarea

analizați ce fel de date și ce volum de date veți avea

ideal este să aruncați un ochi și asupra memoriei și a procesorului

testați dacă sunteți în dubii

ListWrapper & DictWrapper example

```
insert
0.00003890297375619411 seconds - test_data_structure
2.25611674401443451643 seconds - test_data_structure

lookup
0.00004018703475594521 seconds - test_data_structure
0.00006585096707567573 seconds - test_data_structure

delete
0.00007777800783514977 seconds - test_data_structure
0.00004544598050415516 seconds - test_data_structure
```

Paralelism și Concurență – Explicarea conceptelor de paralelism și concurență

Paralelismul, implică execuția simultană a mai multor sarcini sau procese pe mai multe procesoare sau nuclee.

Este vorba despre a face mai multe lucruri în același timp.

Paralelism și Concurență – Explicarea conceptelor de paralelism și concurență

Paralelismul implică faptul că sarcinile se execută independent și în același timp pe unități de procesare separate.

Paralelism și Concurență – Explicarea conceptelor de paralelism și concurență

sarcinile rulează în paralel când rulează în același timp

Paralelism și Concurență - Explicarea conceptelor de paralelism și concurență

Concurența se referă la capacitatea unui sistem de a gestiona mai multe sarcini sau procese în același timp, permițându-le să progreseze fără a aștepta finalizarea reciprocă.

Paralelism și Concurență - Explicarea conceptelor de paralelism și concurență

Concurența nu implică neapărat faptul că sarcinile se execută simultan pe mai multe procesoare sau nuclee. În schimb, ele ar putea să își ia rândul, să împartă resurse și să schimbe contextul.

Paralelism și Concurență – Explicarea conceptelor de paralelism și concurență

sarcinile pot rula în orice ordine: pot fi executate în paralel sau în secvență

Paralelism și Concurență - Explicarea conceptelor de paralelism și concurență

toate sarcinile paralele sunt concurente, dar nu invers

Termenul **secvenţial** (sequential) poate fi folosit în două moduri diferite.

Poate însemna că un anumit set de sarcini trebuie rulat într-o ordine strictă.

Această limitare pe care sistemul o impune ordinii de executare a sarcinilor.

Întreruperile (preemption) se întâmplă atunci când o sarcină este întreruptă (involuntar) pentru ca alta să ruleze.

Paralelism și Concurență – Explicarea conceptelor de paralelism și concurență

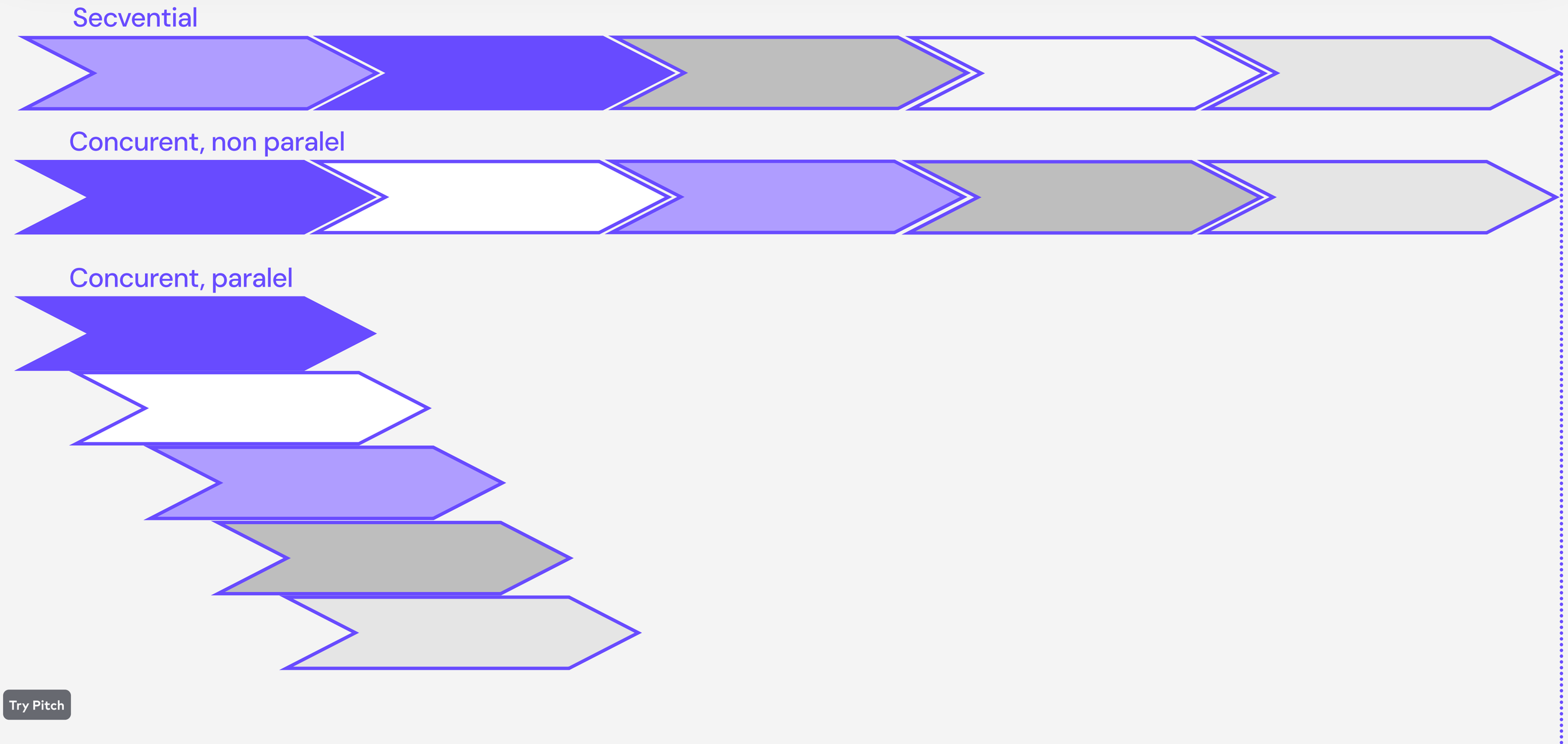
Secvential

Paralelism și Concurență - Explicarea conceptelor de paralelism și concurență

Secvential

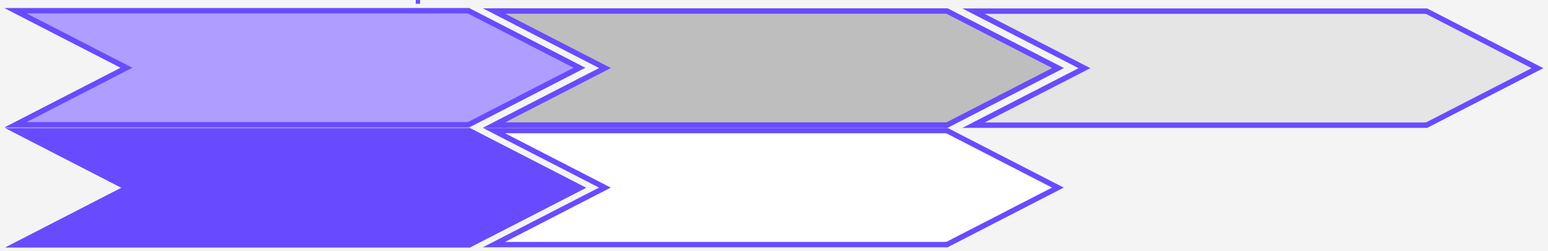
Concurent, non paralel

Paralelism și Concurență – Explicarea conceptelor de paralelism și concurență



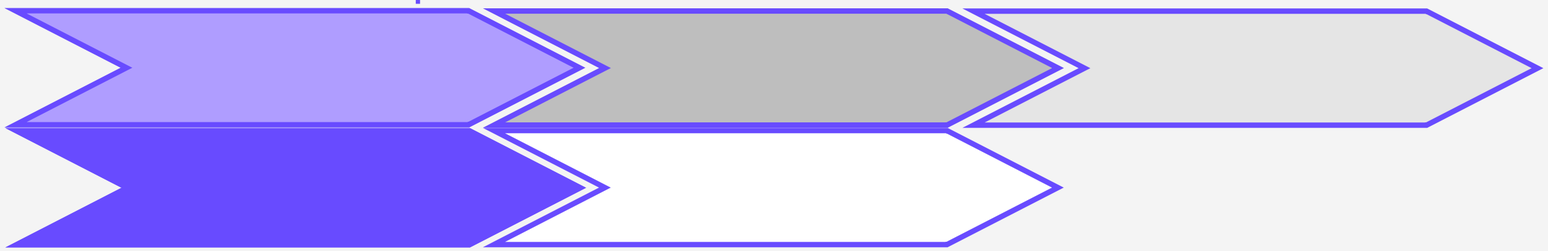
Paralelism și Concurență - Explicarea conceptelor de paralelism și concurență

Paralel cu întreruperi

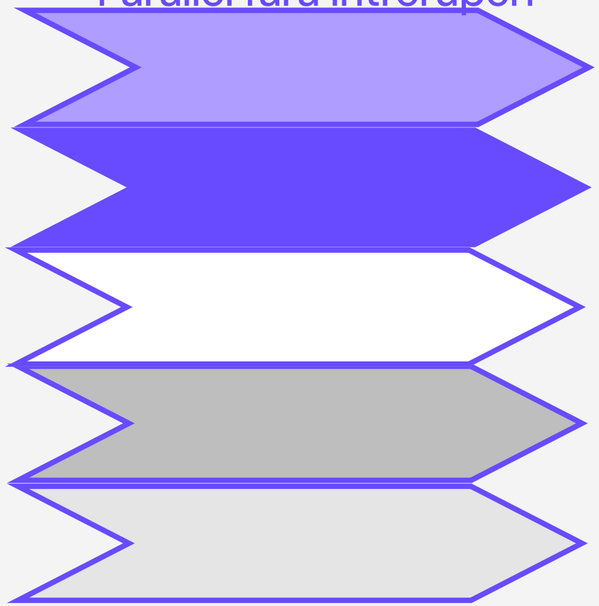


Paralelism și Concurență - Explicarea conceptelor de paralelism și concurență

Paralel cu întreruperi



Parallel fără întreruperi



Paralelism și Concurență - Utilizarea thread-urilor și proceselor în Python

În Python, concurența este ușor de realizat cu firele de execuție, dar nu aduce întotdeauna paralelism real din cauza GIL-ului în CPython.

Pentru paralelism real, în special pentru sarcini CPU-bound, este mai potrivit să folosești multiprocessing sau alte biblioteci specializate care permit crearea de procese separate.

Threading - cazuri de utilizare

potrivit pentru sarcini legate de I/O, cum ar fi cererile de rețea sau operațiile cu fișiere, unde sarcinile petrec mult timp așteptând

permite execuția concurentă în cadrul unui singur proces

oferă partajarea memoriei, ceea ce poate fi benefic pentru comunicarea între firele de execuție

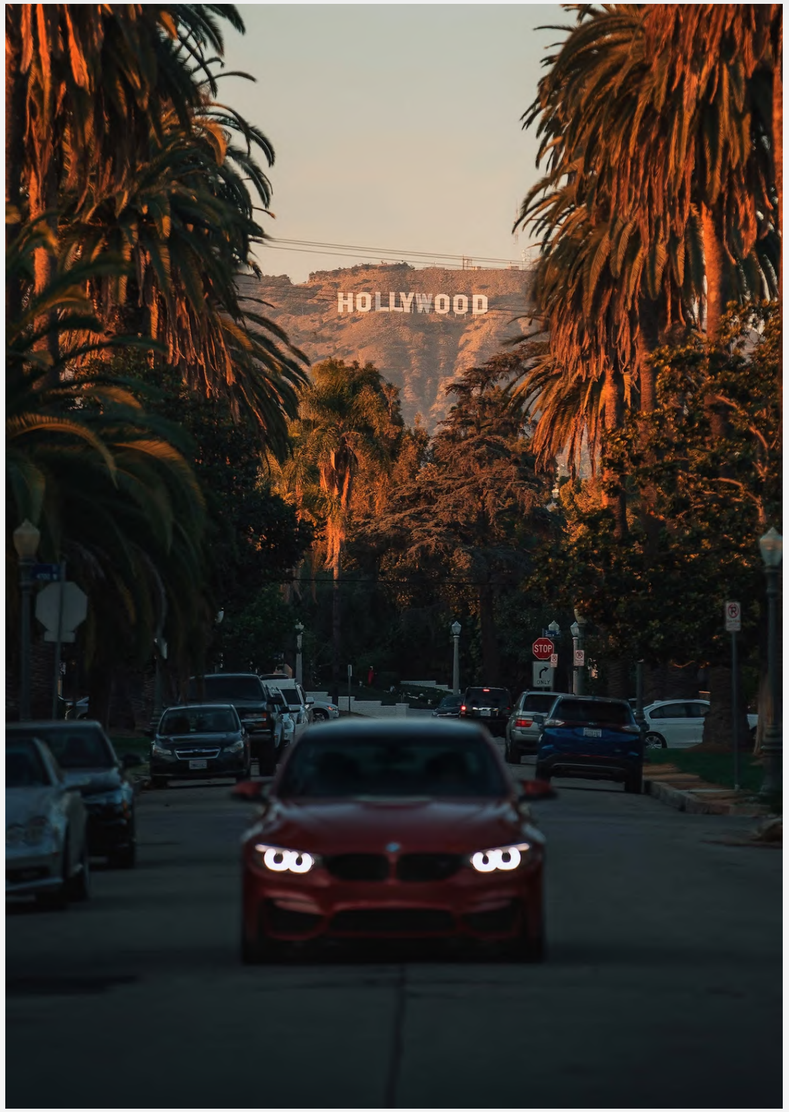
Threading - avantaje

ușor de utilizat și de înțeles

bun pentru gestionarea mai multor sarcini I/O concurente

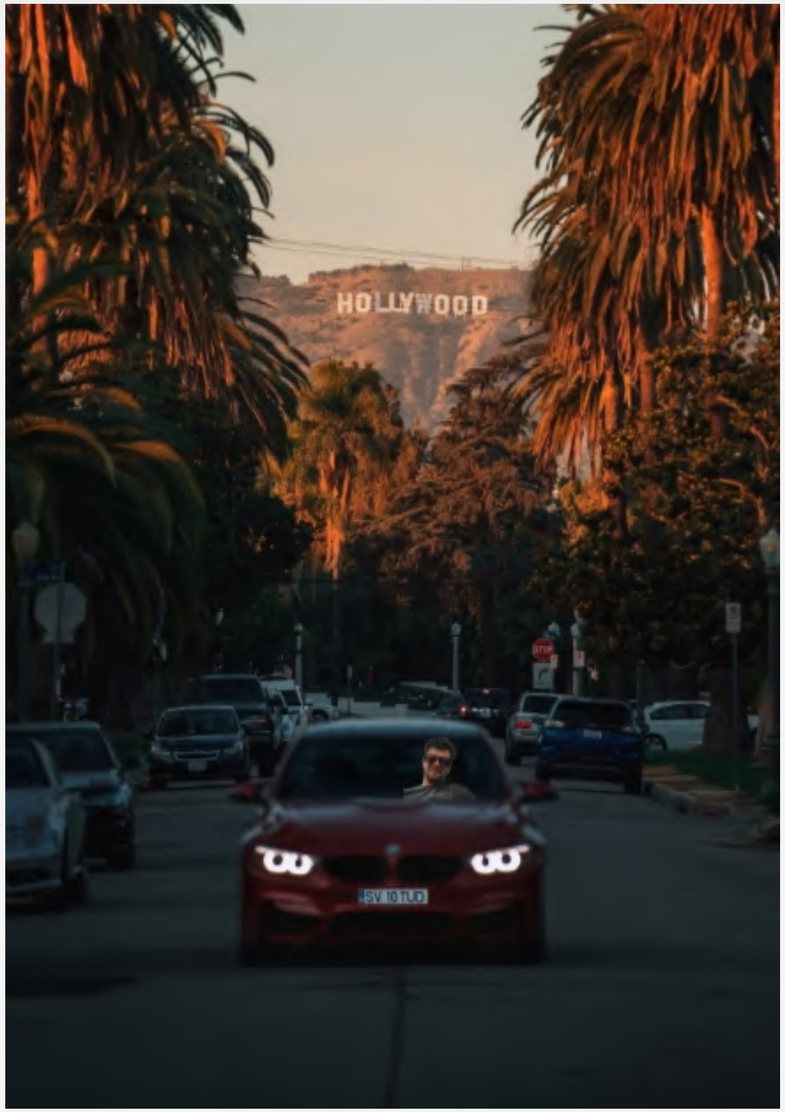
Threading

"Hollywood principle"



Threading

"Hollywood principle"



Threading - dezavantaje

GIL din CPython restricționează adevăratul paralelism pentru sarcinile legate de CPU

necesită sincronizare explicită pentru a evita condițiile de cursă (race conditions) și blocarea reciprocă

Futures - cazuri de utilizare

potrivit pentru sarcini atât I/O-bound, cât și legate de CPU, când dorești o interfață de înalt nivel, prietenoasă pentru utilizator, pentru gestionarea operațiilor asincrone

oferă o modalitate curată de gestionare a rezultatelor sarcinilor asincrone

Futures - avantaje

simplifică programarea asincronă cu o interfață simplă

gestionarea eficientă a sarcinilor concurente

potrivit pentru situațiile în care dorești să inițiezi sarcini și să obții rezultatele ulterior

Futures - dezavantaje

limitat în privința paralelismului pentru sarcini legate de CPU datorită GIL-ului

poate necesita biblioteci externe pentru funcționalități mai avansate

Asyncio - cazuri de utilizare

potrivit pentru sarcini legate de I/O, cum ar fi gestionarea mai multor cereri de rețea, unde există așteptare

permite cod non-blocking, bazat pe evenimente, care poate gestiona eficient mai multe operațiuni concurente

Asyncio - avantaje

oferă adevărata asincronie și non-blocking

excelent pentru aplicații scalabile legate de I/O

evită limitarea dată de GIL

poate gestiona eficient multe sarcini cu un singur ciclu de evenimente (event loop)

Asyncio - dezavantaje

necesită o înțelegere solidă a conceptelor de programare asincronă, precum ciclul de evenimente și corutine

nu este potrivit pentru sarcinile legate de CPU care beneficiază de adevăratul paralelism

Paralelism și Concurență - MapReduce

MapReduce este un model de programare și un "mod" de procesare a datelor folosit pentru a prelucra și genera informații din volume mari de date.

Paralelism și Concurență - MapReduce

Map (Mapper)

Shuffle & Sort

Reduce (Reducer)

Datele sunt prelucrate și transformate dintr-un format brut în perechi cheie-valoare.

O funcție de mapare este aplicată fiecărei înregistrare a datelor și rezultatul va fi o listă de perechi cheie-valoare.

Paralelism și Concurență - MapReduce

Map (Mapper) | **Shuffle & Sort** | Reduce (Reducer)

În această fază, datele rezultate din faza de Map sunt grupate în funcție de cheie.

Toate valorile asociate aceleiași chei sunt colectate împreună pentru a fi procesate ulterior în faza de Reduce.

De asemenea, datele pot fi sortate pentru a simplifica operațiile ulterioare.

Paralelism și Concurență - MapReduce

Map (Mapper) | Shuffle & Sort | **Reduce (Reducer)**

Faza de Reduce constă în aplicarea unei funcții de reducere asupra datelor grupate.

Scopul principal este de a efectua o operație de agregare sau sumare a datelor grupate pentru a genera rezultate finale.

Rezultatele reducer-ului pot fi în cele din urmă stocate sau utilizate în analiza datelor.

Paralelism și Concurență – MapReduce

Map (Mapper)

Shuffle & Sort

Reduce (Reducer)

MapReduce este deosebit de eficient pentru procesarea datelor în medii distribuite sau în sisteme de tip cluster.

Permite paralelizarea și scalabilitatea, ceea ce face ca procesarea unor volume mari de date să fie fezabilă și rapidă.

Paralelism și Concurență - MapReduce

analiza script MapReduce in cele trei moduri discutate

1095695 cuvinte în ~0.00000053597614169121 secunde

Performance Optimization Techniques – Lessons learned

profilarea este primul pas necesar pentru a putea identifica exact unde există deficiențe de performanță

Performance Optimization Techniques – Lessons learned

profilarea este primul pas necesar pentru a putea identifica exact unde există deficiențe de performanță

sistemul intern de profilare al Python este foarte util, dar uneori este dificil de interpretat

Performance Optimization Techniques – Lessons learned

profilarea este primul pas necesar pentru a putea identifica exact unde există deficiențe de performanță

sistemul intern de profilare al Python este foarte util, dar uneori este dificil de interpretat

o complexitate O(1) este mai bună decât O(N) și dacă avem îndoieli cu privire la ceva, există întotdeauna o modalitate mai bună de a face acest lucru.

Performance Optimization Techniques – Lessons learned

Python furnizează structuri de date de bază care pot fi utilizate și folosite necorespunzător pentru a afecta performanța

Performance Optimization Techniques - Lessons learned

Python furnizează structuri de date de bază care pot fi utilizate și folosite necorespunzător pentru a afecta performanța

trebuie să fim conștienți de costul de complexitate al multor operații asupra structurilor de date de bază din Python

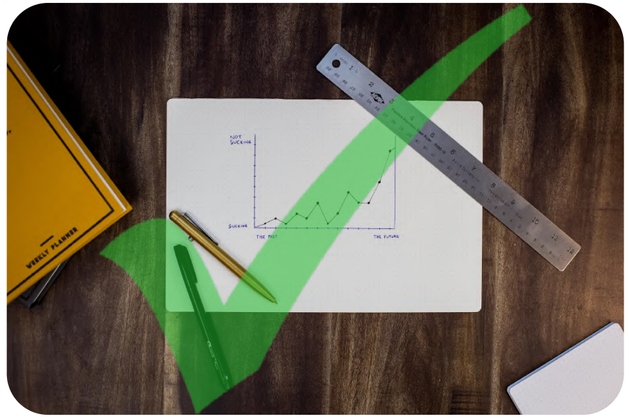
Performance Optimization Techniques - Lessons learned

Python furnizează structuri de date de bază care pot fi utilizate și folosite necorespunzător pentru a afecta performanța

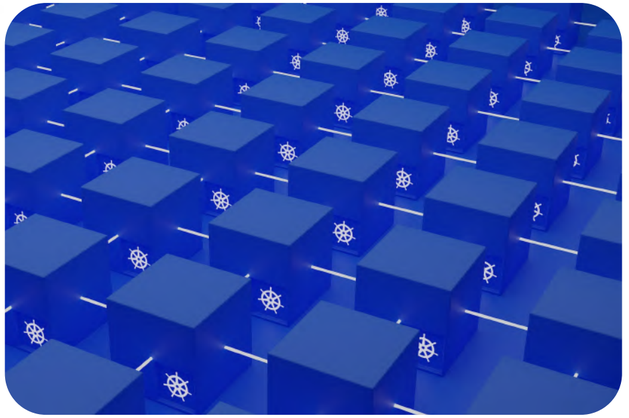
trebuie să fim conștienți de costul de complexitate al multor operații asupra structurilor de date de bază din Python

înțelegerea modului în care funcționează threads, futures și evenimentele asyncio este crucială pentru a crea aplicații eficiente

Boosting Performance with Data Caching



Performance Optimization Techniques



Journey into Microservices