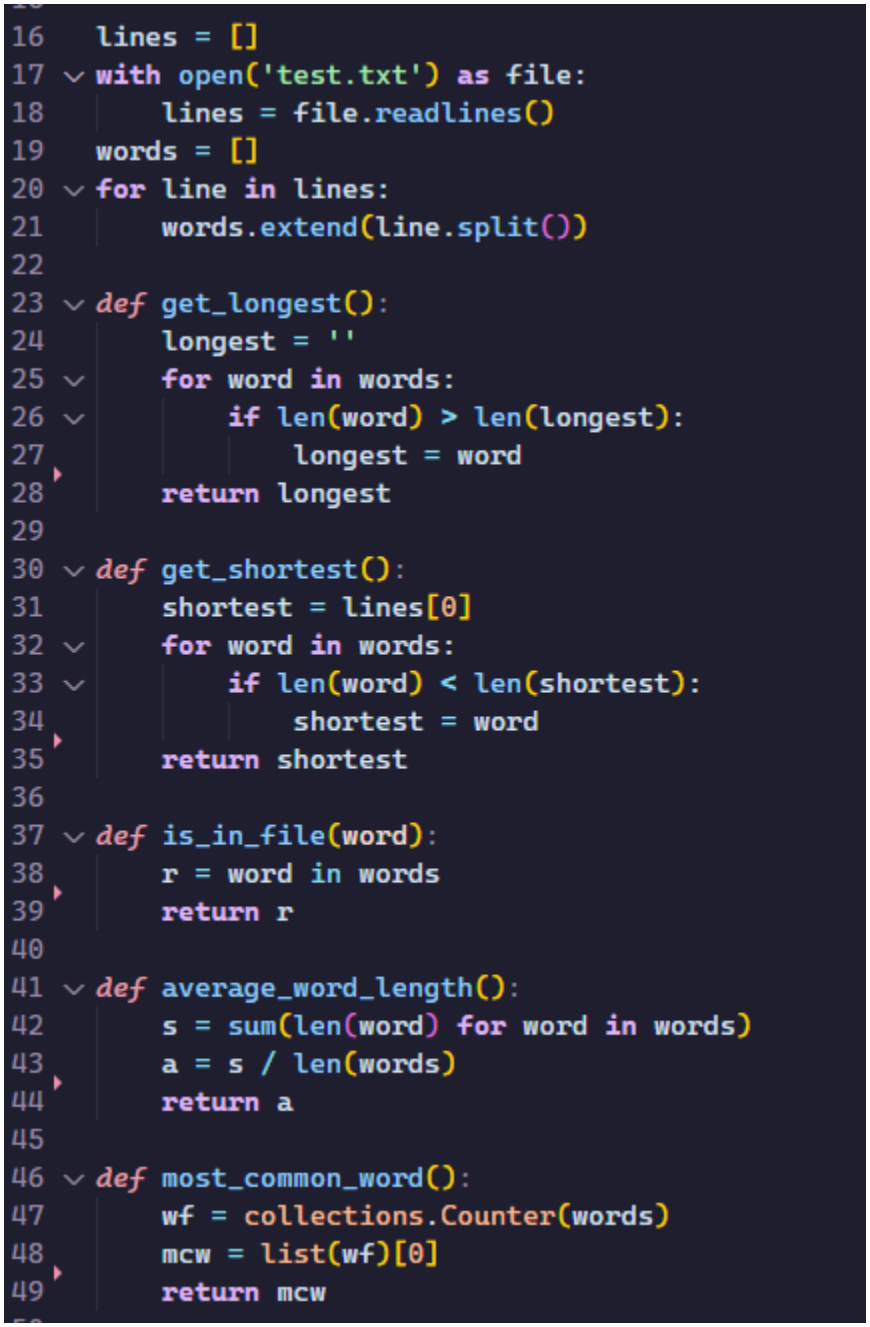
Dokumentálás és mérések

Első program: Szöveg adatok kigyűjtése. Leghosszabb, legrövidebb, leggyakoribb szó, átlagos szó hossz és szó keresés egy fájlban.



```
16  lines = []
17  with open('test.txt') as file:
18      lines = file.readlines()
19  words = []
20  for line in lines:
21      words.extend(line.split())
22
23  def get_longest():
24      longest = ''
25      for word in words:
26          if len(word) > len(longest):
27              longest = word
28      return longest
29
30  def get_shortest():
31      shortest = lines[0]
32      for word in words:
33          if len(word) < len(shortest):
34              shortest = word
35      return shortest
36
37  def is_in_file(word):
38      r = word in words
39      return r
40
41  def average_word_length():
42      s = sum(len(word) for word in words)
43      a = s / len(words)
44      return a
45
46  def most_common_word():
47      wf = collections.Counter(words)
48      mcw = list(wf)[0]
49      return mcw
```

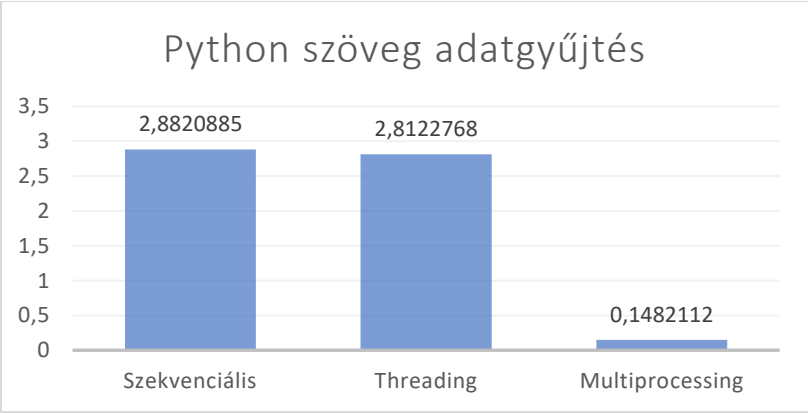
```python
results = []
with concurrent.futures.ThreadPoolExecutor() as executor:
    # results egy future tipusu tomb
    results.append(executor.submit(get_longest))
    results.append(executor.submit(get_shortest))
    results.append(executor.submit(is_in_file, word))
    results.append(executor.submit(average_word_length))
    results.append(executor.submit(most_common_word))
    for r in concurrent.futures.as_completed(results):
        # print(r.result())
        pass
```

Többszálon történő futtatás: A Threadpoolexecutor segítségével létrehozunk egy thread pool-t A thread pool megkönnyíti több thread létrehozását és join-olását. A végrehajtott task-ok egy future objektumot adnak vissza, amelyek később, a task befejezésekor kapnak értéket. A Threadpoolexecutor az executor osztályt egészíti ki, amely rendelkezik 3 függvénnyel, amelyek a threadek kezelésére szolgálnak: submit(), map() és shutdown(). A submit()-nak átadva a függvényt, a Threadpoolexecutor elindítja a szál végrehajtását, és a kapott future objektumot a results listában tároljuk. Miután elindultak a szálak, a concurrent.futures.as_completed() függvénnyel adjuk vissza a future objektumokat, akkor, ha biztosan befejeződött a végrehajtás. A future objektum értékét pedig a result() függvénnyel kapjuk meg.

A (c)Pythonban GIL (Global Interpreter Lock) miatt egy processzoron egyszerre csak egy szál futhat, ezért a szálak menedzselési költsége miatt hosszabb futási idő, mint a szekvenciális változat.

Alternatívaként lehet használni a ProcessPoolExecutor-t, amely szálak helyett processzeket használ. Az öt függvény számára 5 processz-t indítunk és ezzel meglehetősen csökken a program futási ideje.

```python
results = []
with concurrent.futures.ProcessPoolExecutor(max_workers=5) as proc_executor:
    results.append(proc_executor.submit(get_longest))
    results.append(proc_executor.submit(get_shortest))
    results.append(proc_executor.submit(is_in_file, word))
    results.append(proc_executor.submit(average_word_length))
    results.append(proc_executor.submit(most_common_word))
    for r in concurrent.futures.as_completed(results):
        # print(r.result())
        pass
```



Második Python program: Képek letöltése az internetről és grayscale-t szerkeszteni rájuk a pillow, aiohttp és aiofiles modulok segítségével.

Szekvenciális változat:

```python
urls = [
    'https://image.shutterstock.com/image-photo/grey-white-domestic-cat-600w-439615177.jpg',
    'https://image.shutterstock.com/image-photo/gray-kitten-white-background-600w-165838661.jpg',
    'https://image.shutterstock.com/image-photo/lazy-cat-sitting-on-cement-600w-1414420445.jpg',
    'https://image.shutterstock.com/image-photo/siberian-husky-puppy-winter-dog-600w-1871332402.jpg',
    'https://image.shutterstock.com/image-photo/walking-egyptian-mau-on-sunny-600w-1908418108.jpg',
    'https://image.shutterstock.com/image-photo/cute-cat-sitting-on-road-600w-773877655.jpg',
    'https://image.shutterstock.com/image-photo/british-blue-kitten-very-beautiful-600w-796071583.jpg',
    'https://image.shutterstock.com/image-photo/cute-kitten-sits-by-boards-600w-1631858755.jpg'
]

def download_image_seq(url, name):
    img_data = requests.get(url).content
    with open(f'img/{name}', 'wb') as handler:
        handler.write(img_data)

start = time.process_time()

names = []
for i, url in enumerate(urls):
    names.append(url.split('/')[4])
    download_image_seq(url, names[i])

for name in names:
    img = Image.open(f'img/{name}').convert('L')
    img.save(f'img/gray_{name}')

end = time.process_time()

print(f'grayscale szekvencialis ido: {end - start}')
```

A requests modul get függvényével a megadott endpoint url címén lévő adatokat letöltjük és a contents() függvény segítségével a nyers biteket eltároljuk, majd ezt kiírjuk egy jpg fájlba. Ezután a pillow modul Image objektumát használva megnyitjuk és megszerkesztjük a képet, majd visszaírjuk.

A konkurens változat:

```python
async def download_image_async(urls, queue):
    for url in urls:
        name = url.split('/')[4]
        async with aiohttp.ClientSession() as session:
            async with session.get(url) as resp:
                if resp.status == 200: #ok
                    f = await aiofiles.open(f'img/async_{name}', mode='wb')
                    await f.write(await resp.read())
                    await f.close()
                    await queue.put(f'img/async_{name}')


async def grayscale_image(queue):
    while True:
        name = await queue.get()
        img = Image.open(name).convert('L')
        img.save(name)
        queue.task_done()


async def main():
    queue = asyncio.Queue()
    producers = []
    for i in range(0, len(urls), 2):
        producers.append(download_image_async(urls[i:i+2], queue))
    consumers = []
    for _ in range(4):
        consumers.append(asyncio.ensure_future(grayscale_image(queue)))
    await asyncio.gather(*producers,  return_exceptions=True)
    await queue.join()
    for consumer in consumers:
        consumer.cancel()

start = time.process_time()

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()

end = time.process_time()

print(f'grayscale async ido: {end - start}')
```

Az async kulcsszóval deklarálhatunk coroutine-okat és az await kulcsszóval várunk a coroutine-ok befejeződésére. Az await kulcsszóval lehet várni coroutine, task és future objektumokra. Az event loop futtat aszinkron taskokat, callback-eket, hálózati IO operációkat és szubprocesszeket futtat.

Az asyncio.get_event_loop() visszaadja a jelenlegi event loop-ot, ha nem létezik, akkor létrehoz egyet.

loop.run_until_complete() addig fut a megadott main függvény, ameddig az összes future típusú objektum be nem fejeződik.

asyncio.Queue: get() Kiveszi az elemet a sorból.

join() föggvény hívásával blokkol, ameddig az összes elemet ki nem vették a sorból.

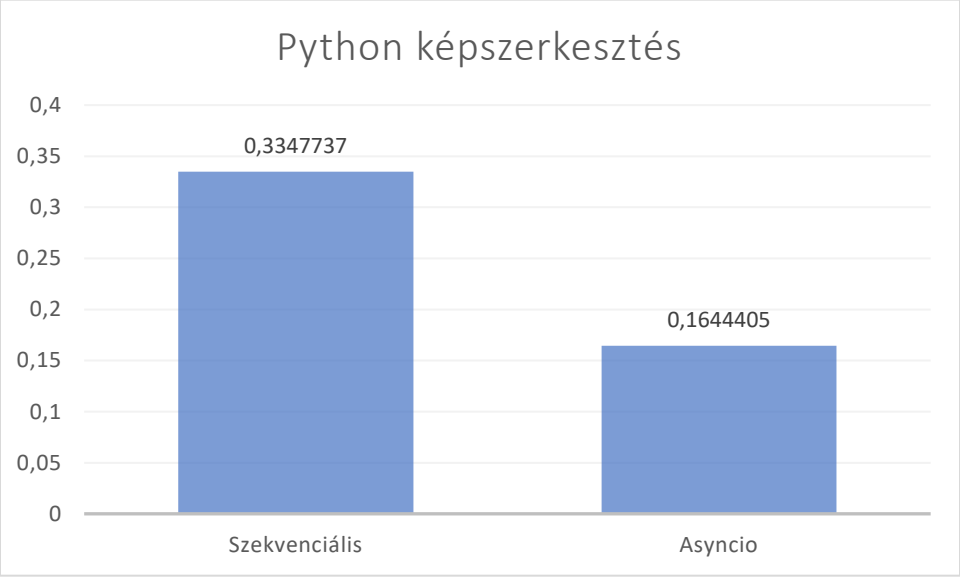
task_done(): Jelzi, hogy a task befejeződött.

producers egy coroutine-okat tartalmazó lista.

asyncio.ensure_future() coroutinok esetén a create_task()- al létrehozza taskokat.

asyncio.gather: Futtatja az összes termelőt, az exception-öket is visszaadja.

Miután a consumerek befejezték a képek szerkesztését le kell őket állítani a cancel() függvénnyel.



Harmadik program: Sudoku rekurzív megoldó probram backtracking algoritmussal.

```
bool find_unassigned(int sudoku[9][9], int *row, int *col) {
    for (*row = 0; *row < 9; (*row)++) {
        for (*col = 0; *col < 9; (*col)++) {
            if (sudoku[*row][*col] == 0) {
                return true;
            }
        }
    }
    return false;
}

int solve(int sudoku[9][9], int level) {
    int row = 0;
    int col = 0;

    if (!find_unassigned(sudoku, &row, &col)) return 1;

    for (int num = 1; num <= 9; num++) {
        if (is_safe(sudoku, row, col, num)) {
#pragma omp task default(none) firstprivate(sudoku, row, col, num, level) shared(start)
            {
                int copy[9][9];
                memcpy(copy, sudoku, 9 * 9 * sizeof(int));
                copy[row][col] = num;
                if (solve(copy, level + 1)) {
                    print_matrix(copy);
                    double end = omp_get_wtime();
                    double time_spent = end - start;
                    printf("\eltelt ido: %lf s\n", time_spent);
                    exit(0);
                }
            }
        }
    }

#pragma omp taskwait
    return 0;
}
```

```
    start = omp_get_wtime();
#pragma omp parallel default(none) shared(sudoku) num_threads(4)
#pragma omp master
    solve(sudoku, 1);
```

A sudoku rekurzív megoldásában a párhuzamosítás a master szálon kezdődik, egy osztott változóval, a sudoku táblával. A solve függvényen belül először meghívunk egy másik függvényt, ami addig megy, amíg nem nullát nem talál a táblában és mellékhatásként módosítja a pointerek által tárolt változók értékét. Ha már nem talált üres mezőt, akkor vége az algoritmusnak. Ha talált üres mezőt, akkor

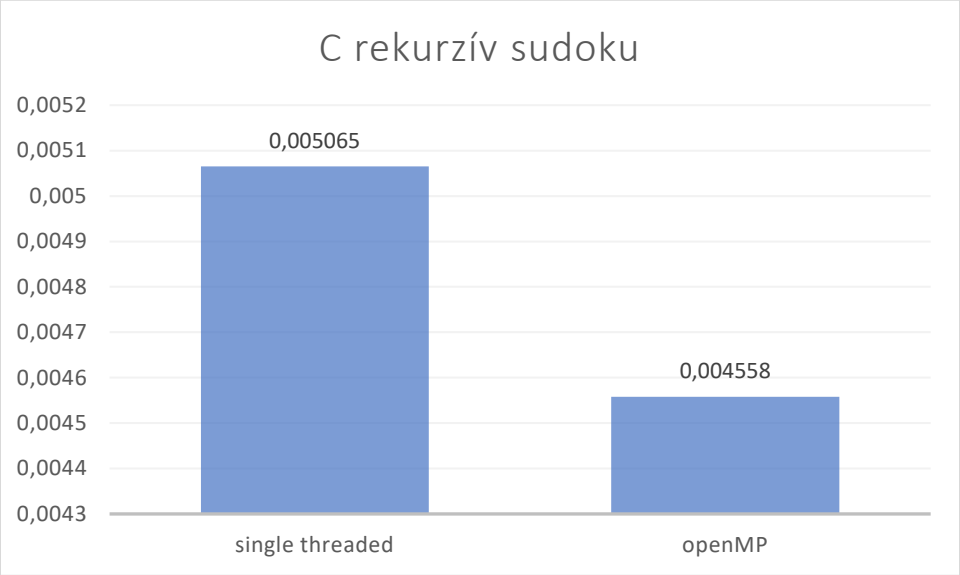
végigmegyünk a számokon 1-től 9-ig és megnézzük, hogy ilyen szám található a sorban, az oszlopban vagy a négyzetben. Ha nincs ilyen szám, akkor létrehoz egy taskot, amely másolatot készít a tábláról az új számot az üres mezőbe helyezve, majd meghívja újra a solve függvényt. Ha kipróbáltuk az összes számot, megvárjuk, hogy az adott szinten indított taskok visszatérjenek és megyünk a következő szintre.

default(none): Kötelezően meg kell adni, hogy milyen kikötések legyenek a változókra.

firstprivate(): A threadnek legyen privát másolata a változóról és inicializálja a korábbi értéket használva.

shared(): Minden thread rendelkezzen a változóval.

taskwait: Megvárja az összes task befejezését mielőtt visszaadná az értéket (Megy a következő mezőre)



Negyedik program: Szó előfordulásának megszámlálása szövegben.

```
const int step =
    (t.len % THREAD_COUNT == 0) ? (t.len / THREAD_COUNT) : (t.len / THREAD_COUNT) + 1;

struct timeval t0, t1, dt;
gettimeofday(&t0, NULL);

int i;
for (i = 0; i < THREAD_COUNT; i++) {
    args *arg = malloc(sizeof(args));
    int *a = malloc(sizeof(int));
    *a = i * step;
    (*arg).start = a;
    (*arg).word = word;
    (*arg).end = *(arg->start) + step;
    pthread_create(&threads[i], NULL, countWord, (void *)arg);
}

int globalCount = 0;
int *returned_value = malloc(sizeof(int));
for (i = 0; i < THREAD_COUNT; i++) {
    pthread_join(threads[i], (void **)&returned_value);
    globalCount += *returned_value;
}
free(returned_value);

printf("Count: %d\n", globalCount);

gettimeofday(&t1, NULL);
timersub(&t1, &t0, &dt);
printf("%ld.%06ld sec\n", dt.tv_sec, dt.tv_usec);

for (i = 0; i < t.len; i++) free(t.arr[i]);

free(t.arr);
return 0;
```

```c
void *countWord(void *arg) {
    args *a = (args *)arg;

    int *count = malloc(sizeof(int));
    *count = 0;
    int increment = strlen(a->word);
    for (int i = *(a->start); i < a->end && i < t.len; i++) {
        for (char *s = t.arr[i]; (s = strstr(s, a->word)); s += increment) {
            printf("%s\n", s);
            sleep(10);
            (*count)++;
        }
    }

    free(a->start);
    free(a);

    return (void *)count;
}
```

A szöveget először beolvassuk egy string tömbbe, majd a tömb mérete alapján felosztjuk a tömböt a szálak számára.