Kurs rozszerzony języka Python

Wykład 6.

Marcin Młotkowski

12 listopada 2024

Plan wykładu

1. Callable objects
2. Wątki
   - Wprowadzenie
   - Wersja wieloprocesorowa
3. Kiedy wątki, kiedy procesy
4. Współpraca między wątkami i procesami
   - Dzielenie się zasobami
   - Zmienne warunkowe
   - Komunikacja między procesami
5. Progamowanie asynchroniczne

Plan wykładu

1. Callable objects
2. Wątki
   - Wprowadzenie
   - Wersja wieloprocesorowa
3. Kiedy wątki, kiedy procesy
4. Współpraca między wątkami i procesami
   - Dzielenie się zasobami
   - Zmienne warunkowe
   - Komunikacja między procesami
5. Progamowanie asynchroniczne

Wszystko jest obiektem.

Wszystko jest obiektem.

A funkcje?

Przykład

```
def foo(x):
    return 2*x

dir(foo)
```

Przykład

```
def foo(x):
    return 2*x

dir(foo)
```

```
['__call__', '__class__', '__closure__', '__code__',
'__init__', '__init_subclass__', '__kwdefaults__',
'__str__', '__subclasshook__', ...]
```

Elementy wykonywalne (ang.*callable*)

Są to te elementy języka Python, które można wywoływać jak funkcję.

Elementy wykonywalne (ang.*callable*)

Są to te elementy języka Python, które można wywoływać jak funkcję.

Przykłady:

- funkcje i metody wbudowane;
- funkcje zdefiniowane przez użytkownika;
- metody obiektu;
- klasy (tworzenie nowego obiektu);
- obiekty implementujące metodę `__call__`.

Przykład obiektu wykonywalnego

```
class Potrojenie:
  def __call__(self, n):
    return self.podwojenie(n) + n

  def podwojenie(self, n):
    return n + n
```

Własny licznik

```
licznik() # zwraca 1
licznik() # zwraca 2
licznik() # zwraca 3
```

Własny licznik

```
licznik() # zwraca 1
licznik() # zwraca 2
licznik() # zwraca 3
```

```
class Licznik:
    def __init__(self):
        self.licznik = 0
    def __call__(self):
        self.licznik += 1
        return self.licznik
```

licznik = Licznik()

Plan wykładu

1. Callable objects
2. Wątki
   - Wprowadzenie
   - Wersja wieloprocesorowa
3. Kiedy wątki, kiedy procesy
4. Współpraca między wątkami i procesami
   - Dzielenie się zasobami
   - Zmienne warunkowe
   - Komunikacja między procesami
5. Progamowanie asynchroniczne

Wstęp

Z Wikipedii:

Wątek (ang. thread) — to jednostka wykonawcza w obrębie jednego procesu, będąca kolejnym ciągiem instrukcji wykonywanym w obrębie tych samych danych (w tej samej przestrzeni adresowej).

Wątki tego samego procesu korzystają ze wspólnego kodu i danych, mają jednak oddzielne stosy.

Po co używać wątków

- zrównoleglenie wolnych operacji wejścia/wyjścia (ściąganie pliku/obsługa interfejsu)
- jednoczesna obsługa wielu operacji, np. serwery WWW

Moduły wątków w Pythonie

- `_thread`: niskopoziomowa biblioteka
- `threading`: wysokopoziomowa biblioteka, korzysta z `_thread`;
- `multiprocessing`
- `concurrent.futures`
- `asyncio`

Jak korzystać z wątków (1. sposób)

```
import threading

class MojWatek(threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        """Operacje wykonywane w wątku"""
        pass
```

Jak korzystać z wątków (2. sposób)

```
from threading import Thread

th1 = Thread(target=Licznik())
th1 = Thread(target=lambda x, y: x * y, args=(2, 3))
```

Przykładowe zadanie

Badanie pierwszości liczby.

Zbadanie podzielności przez kolejne liczby, ale rozdzielimy pracę na różne wątki.

Podział przedziału na podprzedziały

```
def split_domain(m, n, k):
    """Przedział [m, n) dzielimy na k mniej więcej
    równych odcinków"""
    assert m < n
    assert k < n - m

    interwal = (n - m) // k

    wynik = [ (m + interwal * i, m + interwal * (i + 1))
                for i in range(k) ]
    wynik[-1] = (m + interwal * (k - 1), n)
    return wynik
```

Testowanie pierwszości w przedziale

```
def test_prim(k, m, n):
    for i in range(m, n):
        if k % i == 0:
            return False
    return True
```

Rozdzielanie pracy na wątki

strategia 1 ile przedziałów tyle wątków: wątek wykonuje zadanie dla jednego przedziału i kończy pracę;

strategia 2 jest mniej wątków niż zadań, jeden wątek może wykonać wiele zadań.

Implementacja 1. strategii

```
liczba = 2**19 - 1

przedzialy = split_domian(2, liczba, 10)

ths = [ Thread(target=test_prim, args=(liczba, *para)
              for para in przedzialy ]

# wystartowanie watkow
[ th.start() for th in ths ]
```

Implementacja 2. strategii

- tworzymy kolejkę `shared_queue` z zadaniami do wykonania;
- uruchamiamy pewną liczbę wątków, tzw. workerów; które pobierają z kolejki zadania i je wykonują; po wykonaniu czekają na kolejne zadanie.

Zadanie

Zadanie

Jest to krotka (f, arg), gdzie f to pewna funkcja, a *arg* to argumenty tej funkcji.

shared_queue

Obiekt klasy queue.Queue() dedykowany do pracy z wątkami. Podstawowe metody:

- .put(item) włożenie elementu do kolejki;
- .get() pobranie elementu z kolejki lub czekanie aż się coś tam pojawi.

Implementacja workera

```
def worker(shared_queue):

    while True:
        item = shared_queue.get()

        # sygnał końca pracy
        if item is None:
            break

        f, arg = item
        wynik = f(*arg)
        print(f"{f.__name__}({arg}) = {wynik}")

        shared_queue.task_done()
```

Utworzenie i uruchomienie workerów

```
import threading, queue

shared_queue = queue.Queue()

watki = [ threading.Thread(target=worker,
                           args=(shared_queue, ))
              for _ in range(4) ]
[ w.start() for w in watki ]
```

Przygotowanie zadań

```
for pair in split_domain(2, liczba, 10):
    arg = (liczba, *pair)
    shared_queue.put((suma, arg))

# sygnał dla workerów końca pracy
[ shared_queue.put(None) for w in watki ]
```

Gotowe! Prawie;-)

Zbieranie wyników

Słownik na wyniki: kluczem jest zadanie, wartością wynik

```
wyniki = {}
```

Zbieranie wyników

Słownik na wyniki: kluczem jest zadanie, wartością wynik

```
wyniki = {}
```

Przekazanie słownika do workera:

```
watki = [ Thread(target=worker,
                 args=(shared_queue, wyniki))
          for _ in range(4) ]
```

Zbieranie wyników

Słownik na wyniki: kluczem jest zadanie, wartością wynik

```
wyniki = {}
```

Przekazanie słownika do workera:

```
watki = [ Thread(target=worker,
                 args=(shared_queue, wyniki))
          for _ in range(4) ]
```

Przekazanie wyniku w workerze

```
def worker(shared_queue, wyniki):
    ...
    wynik = f(*arg)
    wyniki[(f, arg)] = wynik
    ...
```

Zbieranie wyników

Problem

Musimy poczekać, aż wszystkie wątki skończą pracę, by zbadać zawartość słownika `wyniki`.

Zbieranie wyników

Problem

Musimy poczekać, aż wszystkie wątki skończą pracę, by zbadać zawartość słownika `wyniki`.

Rozwiązanie

```
[ t.join() for t  in ths ]
print(wyniki)
```

Wieloprocesorowość

Jak mamy wiele procesorów, to może da się je wykorzystać:
`multiprocessing`

Zmiany

```
import multiprocessing as mp

shared_queue = mp.JoinableQueue()

watki = [ mp.Process(target=worker,
                     args=(shared_queue, wyniki))
          for _ in range(5) ]
```

Zbieranie wyników

Problem

Każdy proces ma własną pamięć, więc każdy ma własny słownik.

Zbieranie wyników

Problem

Każdy proces ma własną pamięć, więc każdy ma własny słownik.

Rozwiązanie: zarządca pamięci

```
manager = mp.Manager()
wyniki = manager.dict()
```

Proste rozwiązanie

Jeden program, wiele danych.

```
from multiprocessing import Pool

args = [ (liczba, *pair)
             for pair in split_domain(2, liczba, 8)]

with Pool(8) as p:
    wynik = p.map(test_prim, args)
```

Zmienna `wynik` jest wektorem wartości zwróconych przez poszczególne wątki.

Plan wykładu

1. Callable objects
2. Wątki
   - Wprowadzenie
   - Wersja wieloprocesorowa
3. Kiedy wątki, kiedy procesy
4. Współpraca między wątkami i procesami
   - Dzielenie się zasobami
   - Zmienne warunkowe
   - Komunikacja między procesami
5. Progamowanie asynchroniczne

Efektywność wątków: GIL

Efektywność wątków: GIL

Źródło: Wikimedia

Efektywność standardowych wątków

Global Interperter Lock (GIL)

Tylko jeden wątek ma dostęp do bytecodu.

Operacje I/O

GIL jest zwalniany podczas czekania na operacje We/Wy.

Pytanie: To kiedy używać wątków?

Pytanie: To kiedy używać wątków?

Odpowiedź: Jak mamy do czynienia z operacjami we/wy. Czyli: odczyt z dysku, komunikacja sieciowa.

multiprocessing

W przypadku wielu procesów, każdy proces ma swojego GIL'a.

Plan wykładu

1. Callable objects
2. Wątki
   - Wprowadzenie
   - Wersja wieloprocesorowa
3. Kiedy wątki, kiedy procesy
4. Współpraca między wątkami i procesami
   - Dzielenie się zasobami
   - Zmienne warunkowe
   - Komunikacja między procesami
5. Progamowanie asynchroniczne

Jedna zmienna wiele wątków

```
licznik = 0

def worker():
    global licznik
    licznik = licznik + 1
    ...
```

Zagadka

Jaka jest wartość zmiennej `licznik` na końcu programu?

Zagadka

Jaka jest wartość zmiennej `licznik` na końcu programu?

Teoria

Tyle, ile wątków

Zagadka

Jaka jest wartość zmiennej `licznik` na końcu programu?

Teoria

Tyle, ile wątków

Praktyka

Różnie ;-)

Operacje atomowe?

i = i + 1

```
LOADFAST 0
LOAD_CONST 1
BINARY_ADD
STORE_FAST 0
```

Operacje atomowe?

i = i + 1

```
LOADFAST 0
LOAD_CONST 1
BINARY_ADD
STORE_FAST 0
```

i = i + 1

```
LOADFAST 0
LOAD_CONST 1
BINARY_ADD
STORE_FAST 0
```

Blokady

```
licznik = 0
lock = Lock()

def worker():
    global lock
    global licznik

    lock.acquire()
    total_distance = total_distance + 1
    lock.release()
    ...
```

Inne blokady

RLock

Wątek może założyć blokadę dowolną liczbę razy, i tyleż razy musi ją zwolnić. Bardzo spowalnia program.

Semaphore

Blokadę można założyć ustaloną liczbę razy:

```
sem = Semaphore(3)
sem.acquire()
sem.acquire()
sem.acquire()
sem.acquire() # blokada
```

Czekanie na zasób

Jeden wątek (barman) nalewa mleko do szklanki, drugi (klient) czeka na napełnienie szklanki do pełna i wypija mleko.
Szklanka jest pełna, gdy jej wartość jest 5, pusta: 0.

Implementacja picia mleka

```
lck = Lock()
```

Nalewanie:

```
lck.acquire()
for i in range(5):
    szklanka_mleka = szklanka_mleka + 1
lck.release()
```

Wypijanie:

```
while szklanka_mleka != 5: pass #!!!

lck.acquire()
while szklanka_mleka > 0:
    szklanka_mleka = szklanka_mleka - 1
lck.release()
```

Zmienne warunkowe

Mechanizm który pozwala na usypianie i budzenie wątków.

Implementacja

```
lck = threading.Condition()
```

Konsumpcja:

```
lck.acquire()
while szklanka_mleka != 5:
    lck.wait()
while szklanka_mleka > 0:
    szklanka_mleka = szklanka_mleka - 1
lck.release()
```

Nalewanie:

```
lck.acquire()
for i in range(5):
    szklanka_mleka = szklanka_mleka + 1
lck.notify()
lck.release()
```

Zmienne warunkowe

- Zmienne warunkowe są zmiennymi działającymi jak blokady (`aquire()`, `release()`);
- metoda `wait()` zwalnia blokadę i usypia bieżący wątek;
- metoda `notify()` budzi jeden z uśpionych wątków (na tej zmiennej warunkowej), `notifyAll()` budzi wszystkie uśpione wątki.

Wady takiego mechanizmu

- jest tylko jedna szklanka, można do niej tylko nalewać albo tylko z niej pić;
- barman nie może nalać więcej szklanek na zapas i iść do domu

Bezpieczne struktury

Thread-safety

Struktura danych jest *thread-safe*, jeśli może być bezpiecznie używana w środowisku wielowątkowym.

Struktury danych do programów wielowątkowych

Klasa `Queue`:

- Jest to kolejka FIFO, thread–safe;
- Konstruktor: `Queue(`*`rozmiar`*`)`
- pobranie elementu (z usunięciem): `.get()`; gdy kolejka jest pusta zgłasza wyjątek `Empty`
- `.get(True)`: gdy kolejka jest pusta, wątek jest usypiany;
- umieszczenie elementu: `.put(`*`element`*`)`, gdy kolejka jest pełna to zgłaszany jest wyjątek `Full`;
- umieszczenie elementu: `.put(`*`element`*`, True)`, gdy kolejka jest pełna wątek jest usypiany;
- `.full()`, `.empty()`

Warianty klasy Queue

- `LifoQueue`
- `PriorityQueue`

Bar mleczny: inne rozwiązanie

```
def mlekopij(q):
    while True:
        szklanka_mleka = q.get()
        q.task_done()

q = queue.Queue()
m = threading.Thread(target=mlekopij, args=(q,))
m.start()

for mleczko in bar_mleczny:
    q.put(mleczko)

q.join()
m.join()
```

Wymiana informacji między procesami

multiprocessing.Value

```
val = Value("i", 0)
...
val.value = 512
```

Wymiana informacji między procesami

multiprocessing.Value

```
val = Value("i", 0)
...
val.value = 512
```

multiprocessing.Queue

```
q = Queue()
...
q.put(wartosc)
q.get()
```

Komunikacja synchroniczna

```
par_conn, child_conn = Pipe()
...
child_conn.send([1, '''dwa''', 3.0])
...
print(par_conn.recv())
```

Plan wykładu

1. Callable objects
2. Wątki
   - Wprowadzenie
   - Wersja wieloprocesorowa
3. Kiedy wątki, kiedy procesy
4. Współpraca między wątkami i procesami
   - Dzielenie się zasobami
   - Zmienne warunkowe
   - Komunikacja między procesami
5. Progamowanie asynchroniczne

Współprogramy *(ang. coroutines)*

Współprogram

Pewien rodzaj podprogramu (procedury, funkcji etc.), który może zostać zawieszony a sterowanie jest przekazywane do innego współprogramu.

Współprogramy *(ang. coroutines)*

Współprogram

Pewien rodzaj podprogramu (procedury, funkcji etc.), który może zostać zawieszony a sterowanie jest przekazywane do innego współprogramu.

Czasem niektórzy mówią *korutyny*.

Współprogramy w Pythonie

Deklarowanie współprogramu:

```
async def foo():
    ...
    await obiekt
    ...
```

To nie funkcja, a raczej generator.

Zawieszanie działania programu

```
await obiekt
```

Obiekt powinien być *awaitable*. Np. `time.sleep()`.

Uruchomienie

Pojedynczy współprogram

```
import asyncio

asyncio.run(foo())
```

Lista współprogramów

```
import asyncio

wspolprogramy = [<lista współprogramów>]

await asyncio.gather(*wspolprogramy)
```

Wyszukanie słowa Python w plikach

Poszukamy słowa "Python" w plikach synchronicznie i asynchronicznie.

Przeszukiwanie plików

```
def szukanie(path):
    for item in os.listdir(path):
        fullname = os.path.join(path, item)
        if os.path.isfile(fullname):
            find(fullname)
```

Sprawdzenie

```
def find(fname):
    with open(fname, 'rb') as fh:
        content = fh.read()
    if b"python" in content:
        print(f"Znalazłem w {fname} :)")
        return True
    print(f"Nie ma {fname} :(")
    return False

szukanie("/tmp/dane/Pictures")
```

Zbieranie plików

```
async def szukanie(path):
    coros = []
    for item in os.listdir(path):
        fullname = os.path.join(path, item)
        if os.path.isfile(fullname):
            coros.append(find(fullname))
    await asyncio.gather(*coros)

asyncio.run(szukanie("/tmp/dane/Pictures"))
```

Wersja asynchroniczna

```
import aiofiles

async def find(fname):
    async with aiofiles.open(fname, mode='rb') as fh:
        content = await fh.read()
    if b'python' in content:
        print(f"{fname}: znalazłem")
        return True
    print(f"{fname}: nie ma:(")
    return False
```