Raspodijeljeni sustavi (RS)

Nositelj: doc. dr. sc. Nikola Tanković
Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli

(3) Asinkroni Python: Osnove *asyncio* biblioteke

Asinkronost je koncept koji označava mogućnost simultanog izvršavanja više zadataka pri čemu se zadaci izvršavaju neovisno jedan o drugome, odnosno ne čekaju jedan na drugi da se završe, već se odvijaju neovisno o međusobnim vremenskim ograničenjima. U Pythonu, asinkrono programiranje omogućuje nam da zadatke izvršavamo konkurentno, bez blokiranja izvršavanja programa i to bez korištenja tradicionalnih multi-threading tehnika kroz programske dretve. Navedeno je korisno za zadatke poput I/O operacija, mrežnih operacija pozivanjem API-eva, obrade velikih količina podataka, upravljanje podacima i sl. Kroz ovu skriptu naučit ćete pisati asinkroni Python kod koristeći biblioteku asyncio.

Posljednje ažurirano: 21.11.2024.

Sadržaj

1. `asyncio` biblioteka

`asyncio` je biblioteka koja se koristi za pisanje konkurentnog koda koristeći `async/await` sintaksu. Ova biblioteka omogućuje nam da pišemo asinkroni kod koji se izvršava konkurentno, bez blokiranja izvršavanja programa te služi kao svojevrsni **temelj za pisanje asinkronih programa u Pythonu**.

Datoteka je uključena u standardnu biblioteku **Pythona 3.7+** pa ju nije potrebno naknadno instalirati.

Kratki osvrt na paralelno i konkurentno izvršavanje:

Paralelno izvršavanje (*eng. Parallelism*) sastoji se od izvršavanja više operacija simultano, odnosno u isto vrijeme. Ovo se postiže korištenjem prvenstveno više procesnih jedinica (*eng. CPU Cores*). Paralelno izvršavanje je fizičko i odvija se na različitim procesorskim jedinicama.

Konkurentno izvršavanje (*eng. Concurrency*) sastoji se od izvršavanja više operacija u isto vrijeme, ali ne nužno simultano. To znači da se operacije mogu međusobno preklapati u vremenu, ali se izmjenjuju u svom izvršavanju, koristeći najčešće jednu procesorsku jedinicu odnosno iste resurse. Konkurentnost se ostvaruje kroz mehanizme kao što su asinkrono programiranje, višedretvenost (*eng. multithreading*) te programiranje bazirano na događajima (*eng. event-driven programming*).

Na ovom kolegiju dotaknuti ćemo se prvenstveno **konkurentnog izvršavanja** kroz asinkrono programiranje, a u nešto manjoj mjeri i na paralelno izvršavanje.

1.1. Korutine (eng. Coroutines)

Ključne riječi `async` i `await` koriste se za:

1. **definiranje asinkronih** (`async`) **funkcija** (koje vraćaju `coroutine` objekte) te za
2. **čekanje na rezultat izvršavanja asinkronih funkcija** (`await`).

Kako bismo simulirali asinkrono izvršavanje, iskoristit ćemo funkciju `asyncio.sleep()` koja simulira čekanje određenog vremena zadanog u **sekundama**.

Sintaksa:

```
asyncio.sleep(delay)
```

- `delay` - broj sekundi koliko želimo čekati - odgoditi izvršavanje koda

```
import asyncio

async def main():
    print('Hello')
    await asyncio.sleep(1)
    print('World')
```

U gornjem primjeru, funkcija `main()` je asinkrona funkcija koja ispisuje "Hello", čeka 1 sekundu te ispisuje "World". Međutim, kako bi se funkcija `main()` izvršila, potrebno ju je pokrenuti pomoću `asyncio.run()` funkcije.

`asyncio.run()` je također funkcija iz `asyncio` biblioteke kojom pokrećemo asinkronu (**korutinu**) pokretanjem tzv. *event loopa*. Kao obavezan argument, prima asinkronu funkciju koju želimo pokrenuti - u ovom slučaju to je funkcija `main()`.

Sintaksa:

```
asyncio.run(coroutine)
```

- `coroutine` - asinkrona funkcija koju želimo pokrenuti

Event loop je mehanizam koji upravlja izvršavanjem asinkronih funkcija, odnosno **korutina**.

Korutina (*eng. coroutine*) je specifična vrsta funkcije koja se može zaustaviti i nastaviti izvršavanje u bilo kojem trenutku. Korutine se koriste za pisanje asinkronog koda u Pythonu.

```
import asyncio

async def main():
    print('Hello')
    await asyncio.sleep(1)
    print('World')

asyncio.run(main())
```

Izvršavanjem gornjeg koda, dobit ćemo ispis:

```
Hello
World
```

Kao što vidimo, ispis `"Hello"` se pojavljuje odmah, dok se ispis `"World"` pojavljuje nakon 1 sekunde. Na ovaj način, napisali smo najjednostavniji primjer asinkronog izvršavanja koda.

1.2 Konkurentno izvršavanje više korutina

Recimo da imamo više korutina koje želimo pokrenuti. U praksi ćemo htjeti logiku za dohvaćanje podataka s weba (npr. preko API-ja) odvojiti od logike za obradu tih podataka. Idemo simulirati takav primjer:

```
import asyncio

async def fetch_data(): # primjer jednostavne korutine koja simulira dohvaćanje podataka
    print('Dohvaćam podatke...')
    data = {'iznos': '3000', 'stanje': 'uspješno'}
    await asyncio.sleep(2)
    print('Podaci dohvaćeni.')
    return data

async def main():
    data = await fetch_data()
    print(f'Podaci: {data}')

asyncio.run(main())
```

Što će se dogoditi kada pokrenemo gornji kod?

▶ Spoiler alert! Odgovor na pitanje

Međutim, što ako imamo više asinkronih funkcija koje želimo pokrenuti, a koje imaju **različite duljine trajanja/izvođenja**? U praksi to može biti slučaj kada dohvaćamo podatke s više različitih API-ja, gdje su neki API-evi brži, a neki sporiji.

Idemo simulirati takav primjer.

```
import asyncio

async def fetch_api_1():
    print('Dohvaćam podatke s API-ja 1...')
    await asyncio.sleep(2)
    print('Podaci s API-ja 1 dohvaćeni.')
    return {'api_1': 'uspješno'}

async def fetch_api_2():
    print('Dohvaćam podatke s API-ja 2...')
    await asyncio.sleep(4)
    print('Podaci s API-ja 2 dohvaćeni.')
    return {'api_2': 'uspješno'}
```

Kako ćemo definirati funkciju `main()` koja će pokrenuti obje asinkrone funkcije `fetch_api_1()` i `fetch_api_2()`?

Možemo pokušati na sljedeći način:

```python
async def main():
    podaci_1 = await fetch_api_1()
    podaci_2 = await fetch_api_2()

    print(f'Podaci s API-ja 1: {podaci_1}')
    print(f'Podaci s API-ja 2: {podaci_2}')

asyncio.run(main())
```

Pokrenite kod, koliko je vremena potrebno da se dohvate svi podaci? Zašto?

▶ Spoiler alert! Odgovor na pitanje

Kako bismo riješili ovaj problem, koristit ćemo funkciju `asyncio.gather()` koja omogućuje pokretanje **više korutina konkurentno**. Ova funkcija prima više asinkronih funkcija kao argumente te ih pokreće istovremeno (ne nužno paralelno, ali konkurentno).

Sintaksa:

```python
asyncio.gather(*coros)
```

- `*coros` - argumenti su asinkrone funkcije koje želimo pokrenuti

```python
async def main():
    podaci_1, podaci_2 = await asyncio.gather(fetch_api_1(), fetch_api_2())

    print(f'Podaci s API-ja 1: {podaci_1}')
    print(f'Podaci s API-ja 2: {podaci_2}')
```

Pokrenite kod, koliko je vremena potrebno da se dohvate svi podaci? Zašto?

▶ Spoiler alert! Odgovor na pitanje

Primjer: Definirat ćemo korutinu `timer()` koja će simulirati otkucaje timera svake sekunde. Korutina prima 2 argumenta: naziv timera i broj sekundi koliko će trajati, a zatim svake sekunde ispisuje preostale vrijeme.

```python
import asyncio

async def timer(name, delay):
    for i in range(delay, 0, -1):
        print(f'{name}: {i} sekundi preostalo...')
        await asyncio.sleep(1)
    print(f'{name}: Vrijeme je isteklo!')

async def main():
    await asyncio.gather( # pokrećemo dvije korutine konkurentno
        timer('Timer 1', 3),
        timer('Timer 2', 5)
```

```
    )

asyncio.run(main())
```

- Pokrenite kod i provjerite ispis.

Rezultat:

```
Timer 1: 3 sekundi preostalo...
Timer 2: 5 sekundi preostalo...
Timer 1: 2 sekundi preostalo...
Timer 2: 4 sekundi preostalo...
Timer 1: 1 sekundi preostalo...
Timer 2: 3 sekundi preostalo...
Timer 1: Vrijeme je isteklo!
Timer 2: 2 sekundi preostalo...
Timer 2: 1 sekundi preostalo...
Timer 2: Vrijeme je isteklo!
```

1.3 `asyncio tasks`

Radni zadatak, odnosno `task` u `asyncio` su temeljni gradivni blokovi asinkronog programiranja u Pythonu. `Task` predstavlja izvršnu jedinicu, odnosno asinkronu operaciju, koja se zakazuje (*eng. schedules*) za izvršavanje u `event loop`-u.

`asyncio.create_task()` je funkcija koja stvara novi `Task` objekt koji izvršava asinkronu funkciju. Ova funkcija je korisna kada želimo definirati korutinu koju ćemo zakazati za konkurentno izvršavanje kasnije u programu.

Sintaksa:

```
asyncio.create_task(coroutine)
```

- `coroutine` - asinkrona funkcija koju želimo zakazati za konkurentno izvršavanje
- vraća `Task` objekt (`<class '_asyncio.Task'>`)

Implementirat ćemo prethodne primjer pozivanja API-ja koristeći `asyncio.create_task()`.

```
import asyncio

async def fetch_api_1():
    print('Dohvaćam podatke s API-ja 1...')
    await asyncio.sleep(2)
    print('Podaci s API-ja 1 dohvaćeni.')
    return {'api_1': 'uspješno'}

async def fetch_api_2():
    print('Dohvaćam podatke s API-ja 2...')
    await asyncio.sleep(4)
    print('Podaci s API-ja 2 dohvaćeni.')
    return {'api_2': 'uspješno'}
```

Korutine `fetch_api_1()` i `fetch_api_2()` su iste kao i prije, ali **postoji razlika u načinu pozivanja korutina**.

```
async def main():
    task_1 = asyncio.create_task(fetch_api_1())
    task_2 = asyncio.create_task(fetch_api_2())

    podaci_1 = await task_1
    podaci_2 = await task_2

    print(f'Podaci s API-ja 1: {podaci_1}')
    print(f'Podaci s API-ja 2: {podaci_2}')

asyncio.run(main())
```

Pokrenite kod, koliko je vremena potrebno da se dohvate svi podaci? Zašto?

▶ Spoiler alert! Odgovor na pitanje

Općenito, koristeći `asyncio.create_task()` možemo pokrenuti više korutina konkurentno, a zatim čekati na njihov završetak.

Sintaksa:

```
task_1 = asyncio.create_task(coroutine_1())
task_2 = asyncio.create_task(coroutine_2())

await task_1 # čekamo na završetak prve korutine
await task_2 # čekamo na završetak druge korutine
```

> Dakle, kod iznad će se izvršiti **konkurentno**, a ne sekvencijalno.

1.3.1 Konkurentno izvođenje kroz `asyncio.gather()` i `asyncio.create_task()`

Kombinirajmo prethodne primjere korištenjem `asyncio.create_task()` i `asyncio.gather()`.

Želimo definirati jednu korutinu `korutina(n)` koja će čekati jednu sekundu, a zatim vratiti poruku o završetku izvođenja.

```
import asyncio

async def korutina(n):
    await asyncio.sleep(1)
    return f'Korutina {n} je završila.'
```

U `main()` funkciji ćemo pohraniti 5 korutina u liste `tasks`. Drugim riječima, želimo pohraniti 5 `Task` objekata koji će izvršavati korutine `korutina(n)`, za `n` od 1 do 5.

```
async def main():
  tasks = []

  for i in range(1, 6):
      task = asyncio.create_task(korutina(i))
      tasks.append(task)

  print(tasks) # ispis svih referenci na Task objekte

asyncio.run(main())
```

Kako ovo možemo napraviti elegantnije? `list comprehension` nam može pomoći.

```
async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]
    print(tasks) # ispis svih referenci na Task objekte

asyncio.run(main())
```

Za pokretanje svih korutina konkurentno, ne želimo pisati `await task` za svaki `Task` objekt.

Dakle, **sljedeće nije najbolje rješenje**:

```
async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]

    for task in tasks:
        await task

    print('Sve korutine su završile.')

asyncio.run(main())
```

Zašto? Nigdje ne pohranjujemo rezultate korutina, već samo čekamo na njihov završetak.

Stvari možemo riješiti ovako:

```
async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]

    results = []

    for task in tasks:
        results.append(await task) # čekamo na završetak svake korutine i pohranjujemo rezultat

    print(results)

asyncio.run(main())
```

Međutim, puno bolje rješenje je koristiti `asyncio.gather()`.

- `asyncio.gather()` osim može korutina može primiti i `Task` objekte
- možemo proslijediti jedan ili više `Task` objekata na isti način kao i korutine: `await asyncio.gather(task_1, task_2, task_3)`
- međutim, možemo proslijediti i listu korutina ili `Task` objekata s operatorom `*`: `await asyncio.gather(*tasks)`

```
async def main():
    tasks = [asyncio.create_task(korutina(i)) for i in range(1, 6)]
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
# Ispisuje: ['Korutina 1 je završila.', 'Korutina 2 je završila.', 'Korutina 3 je završila.', 'Korutina 4 je završila.', 'Korutina 5 je završila.']
```

Na ovaj način, `asyncio.gather(*tasks)` čeka na završetak svih korutina i vraća **listu rezultata izvođenja korutina**.

Pogledat ćemo još nekoliko jednostavnih primjera i mjeriti vrijeme izvođenja programa koristeći `time` modul.

Primjer: Definirat ćemo korutinu koja će nakon određenog vremena ispisati poruku.

```
import asyncio
import time

async def kaži_nakon(delay, poruka):
    await asyncio.sleep(delay)
    print(poruka)

async def main():
  print (f"Početak: {time.strftime('%X')}")

  await kaži_nakon(1, 'Pozdraaav!')
  await kaži_nakon(2, 'Kako si?')

  print (f"Kraj: {time.strftime('%X')}")

asyncio.run(main())
```

- Ako pokrenemo program u ovom obliku u 11:00:00, što će biti ispisano?

```
Početak: 11:00:00
Pozdraaav!
Kako si?
Kraj: 11:00:03
```

- Isto možemo pretočiti u Task objekte:

```
async def main():
  print (f"Početak: {time.strftime('%X')}")

  task1 = asyncio.create_task(kaži_nakon(1, 'Pozdraaav!'))
  task2 = asyncio.create_task(kaži_nakon(2, 'Kako si?'))

  await task1
  await task2

  print (f"Kraj: {time.strftime('%X')}")

asyncio.run(main())
```

- ili koristeći `asyncio.gather()`:

```
async def main():
  print (f"Početak: {time.strftime('%X')}")

  task1 = asyncio.create_task(kaži_nakon(1, 'Pozdraaav!'))
  task2 = asyncio.create_task(kaži_nakon(2, 'Kako si?'))

  await asyncio.gather(task1, task2)

  print (f"Kraj: {time.strftime('%X')}")

asyncio.run(main())
```

Primjer: Idemo vidjeti kako možemo na isti način koristiti `asyncio.gather()` za pozivanje prethodne korutine `Timer(name, delay)` koja simulira otkucaje timera svake sekunde. Korutinu želimo pokrenuti 3 puta s različitim vremenima trajanja. Potrebno je definirati `Task` objekte i pohraniti ih u listu `tasks`, a zatim koristiti `asyncio.gather()` za pokretanje svih korutina konkurentno.

```
import asyncio

async def timer(name, delay):
    for i in range(delay, 0, -1):
        print(f'{name}: {i} sekundi preostalo...')
        await asyncio.sleep(1)
    print(f'{name}: Vrijeme je isteklo!')

async def main():
    timers = [
        asyncio.create_task(timer('Timer 1', 3)),
        asyncio.create_task(timer('Timer 2', 5)),
        asyncio.create_task(timer('Timer 3', 7))
    ]

    await asyncio.gather(*timers)

asyncio.run(main())
```

2. Zadaci za vježbu - Korutine, Task objekti, asyncio.gather()

1. **Definirajte korutinu koja će simulirati dohvaćanje podataka s weba**. Podaci neka budu lista brojeva od 1 do 10 koju ćete vratiti nakon 3 sekunde. Listu brojeva definirajte comprehensionom. Nakon isteka vremena, u korutinu ispišite poruku "Podaci dohvaćeni." i vratite podatke. Riješite bez korištenja `asyncio.gather()` i `asyncio.create_task()` funkcija.

2. **Definirajte dvije korutine koje će simulirati dohvaćanje podataka s weba**. Prva korutina neka vrati listu proizvoljnih rječnika (npr. koji reprezentiraju podatke o korisnicima) nakon 3 sekunde, a druga korutina neka vrati listu proizvoljnih rječnika (npr. koji reprezentiraju podatke o proizvodima) nakon 5 sekundi. Korutine pozovite konkurentno korištenjem `asyncio.gather()` i ispišite rezultate. Program se mora izvršavati ~5 sekundi.

3. **Definirajte korutinu `autentifikacija()` koja će simulirati autentifikaciju korisnika na poslužiteljskoj strani**. Korutina kao ulazni parametar prima rječnik koji opisuje korisnika, a sastoji se od ključeva `korisnicko_ime`, `email` i `lozinka`. Unutar korutine simulirajte provjeru korisničkog imena na način da ćete provjeriti nalaze li se par `korisnicko_ime` i `email` u bazi korisnika. Ova provjera traje 3 sekunde.

```
baza_korisnika = [
  {'korisnicko_ime': 'mirko123', 'email': 'mirko123@gmail.com'},
  {'korisnicko_ime': 'ana_anic', 'email': 'aanic@gmail.com'},
  {'korisnicko_ime': 'maja_0x', 'email': 'majaaaaa@gmail.com'},
  {'korisnicko_ime': 'zdeslav032', 'email': 'deso032@gmail.com'}
]
```

Ako se korisnik ne nalazi u bazi, vratite poruku `"Korisnik {korisnik} nije pronađen."`

Ako se korisnik nalazi u bazi, potrebno je pozvati vanjsku korutinu `autorizacija()` koja će simulirati autorizaciju korisnika u trajanju od 2 sekunde. Funkcija kao ulazni parametar prima rječnik korisnika iz baze i lozinku proslijeđenu iz korutine `autentifikacija()`. Autorizacija simulira dekripciju lozinke (samo provjerite podudaranje stringova) i provjeru s lozinkom iz baze_lozinka. Ako su lozinke jednake, korutine vraća poruku `"Korisnik {korisnik}: Autorizacija uspješna."`, a u suprotnom `"Korisnik {korisnik}: Autorizacija neuspješna."`.

```
baza_lozinka = [
  {'korisnicko_ime': 'mirko123', 'lozinka': 'lozinka123'},
  {'korisnicko_ime': 'ana_anic', 'lozinka': 'super_teska_lozinka'},
  {'korisnicko_ime': 'maja_0x', 'lozinka': 's324SDFfdsj234'},
  {'korisnicko_ime': 'zdeslav032', 'lozinka': 'deso123'}
]
```

Korutinu `autentifikacija()` pozovite u `main()` funkciji s proizvoljnim korisnikom i lozinkom.

4. **Definirajte korutinu `provjeri_parnost` koja će simulirati "super zahtjevnu operaciju" provjere parnosti** broja putem vanjskog API-ja. Korutina prima kao argument broj za koji treba provjeriti parnost, a vraća poruku `"Broj {broj} je paran."` ili `"Broj {broj} je neparan."` nakon 2 sekunde. Unutar `main` funkcije definirajte listu 10 nasumičnih brojeva u rasponu od 1 do 100 (koristite `random` modul). Listu brojeva izgradite kroz list comprehension sintaksu. Nakon toga, pohranite u listu `zadaci` 10 `Task` objekata koji će izvršavati korutinu `provjeri_parnost` za svaki broj iz liste (također kroz list comprehension). Na kraju, koristeći `asyncio.gather()`, pokrenite sve korutine konkurentno i ispišite rezultate.

5. **Definirajte korutinu `secure_data` koja će simulirati enkripciju osjetljivih podataka**. Kako se u praksi enkripcija radi na poslužiteljskoj strani, korutina će simulirati enkripciju podataka u trajanju od 3 sekunde. Korutina prima kao argument rječnik osjetljivih podataka koji se sastoji od ključeva `prezime`, `broj_kartice` i `CVV`. Definirajte listu s 3 rječnika osjetljivih podataka. Pohranite u listu `zadaci` kao u prethodnom zadatku te pozovite zadatke koristeći `asyncio.gather()`. Korutina `secure_data` mora za svaki rječnik vratiti novi rječnik u obliku: `{'prezime':` prezime `, 'broj_kartice': 'enkriptirano', 'CVV': 'enkriptirano'}`. Za fake enkripciju koristite funkciju `hash(str)` koja samo vraća hash vrijednost ulaznog stringa.