PPaDS MMXX

Matus Jokay, C-503, matus.jokay@stuba.sk
Roderik Ploszek, C-512, roderik.ploszek@stuba.sk

uim.fei.stuba.sk/predmet/i-ppds

konzultacie dohodou

Brad Solomon

Async IO in Python: A Complete Walkthrough

https://realpython.com/async-io-python

https://realpython.com/courses/python-3-concurrency-asyncio-module

Konkurentne programovanie

Konkurentne programovanie

- Konkurencia je jav, ked pohladom na zdrojovy kod nevieme urcit, v akom poradi sa vykonavaju jednotlive casti kodu

Konkurentne programovanie

- Konkurencia je jav, ked pohladom na zdrojovy kod nevieme urcit, v akom poradi sa vykonavaju jednotlive casti kodu

- Pri jednovlaknovom programe sme to doteraz mali jednoduche: vzdy sme uvazovali nad tym, ze pri toku vykonavania riadenom zasobnikom nemoze ku konkurencii dochadzat

Konkurentne programovanie

- Konkurencia zavadza moznost subezneho (paralelneho) vykonavania; nie je to vsak to iste!

Konkurentne programovanie

- Konkurencia zavadza moznost subezneho (paralelneho) vykonavania; nie je to vsak to iste!

- Pri konkurencii nas zaujima to, ze nevieme poradie vykonanych operacii

Konkurentne programovanie

- Ake moznosti konkurentneho vykonavania programov nam prinasa uroven OS?

Konkurentne programovanie

- Ake moznosti konkurentneho vykonavania programov nam prinasa uroven OS?

- Procesy
- Vlakna

Konkurentne programovanie

- Procesy

- Preco konkurencia?

Konkurentne programovanie

- Procesy

- Preco konkurencia?
- Co moznost paralelneho vykonavania?

Konkurentne programovanie

- Vlakna

- Preco konkurencia?

Konkurentne programovanie

- Vlakna

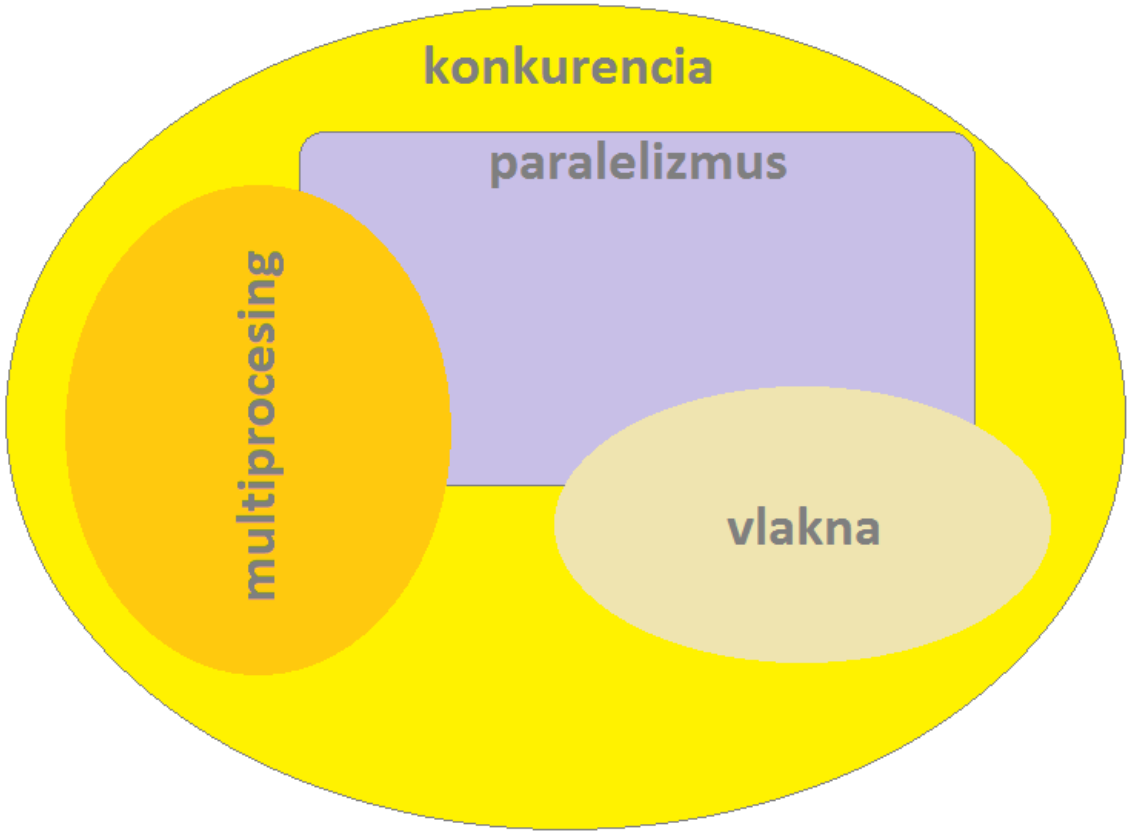
- Preco konkurencia?
- Co moznost paralelneho vykonavania?

Konkurentne programovanie

- Zhrnutie

- Konkurencia
- Paralelizmus

- Multiprocesing
- Vlakna



Konkurentne programovanie

- Ako je to v jazyku Python v3.8?

Konkurentne programovanie

- Ako je to v jazyku Python v3.8?
- Nastroje paralelizmu
- Nastroje konkurencie

Konkurentne programovanie

- Ako je to v jazyku Python v3.8?

- Nastroje paralelizmu
  - Procesy

- Nastroje konkurencie
  - Procesy

Konkurentne programovanie

- Ako je to v jazyku Python v3.8?

- Nastroje paralelizmu
  - Procesy

- Nastroje konkurencie
  - Procesy
  - Vlakna
  - Rozsirene generatory, korutiny

Konkurentne programovanie

- Aby sme boli presni, CPython kvoli GIL nepodporuje paralelny beh vlakien takmer vobec

Konkurentne programovanie

- Aby sme boli presni, CPython kvoli GIL nepodporuje paralelny beh vlakien takmer vobec

- Ine implementacie (Jython, Cython) tento problem nemaju (nemaju totiz GIL) ;)

Konkurentne programovanie

- Dosledok: mozne nepredvidane spravanie kodu!

Konkurentne programovanie

- Dosledok: mozne nepredvidane spravanie kodu!

- Riesenie: spravne riesenie konkurentneho pristupu

Konkurentne programovanie

- Dosledok: mozne nepredvidane spravanie kodu!
- Riesenie: spravne riesenie konkurentneho pristupu
- Otazka: staci riesit konkurenciu, paralelizmus netreba?

Konkurentne programovanie

- Ako sa dosahuje konkurentnost?

Konkurentne programovanie

- Ako sa dosahuje konkurentnost?

- Preemptivny versus kooperativny multitasking

Konkurentne programovanie

- Ako sa dosahuje konkurentnost?

- Preemptivny versus kooperativny multitasking
  - Preemptivny – CPU sa ulohe (task) prideluje transparentne (t.j. uloha o tom “nevie”)
  - Kooperativny – uloha dostane k dispozicii CPU a sama rozhodne, v ktorom okamihu sa ho vzda

Konkurentne programovanie

- Aj preemptivny, aj kooperativny multitasking iba **budia dojem** subezneho vykonavania uloh!

Konkurentne programovanie

- Aj preemptivny, aj kooperativny multitasking iba **budia dojem** subezneho vykonavania uloh!

- Ako?

Konkurentne programovanie

- Aj preemptivny, aj kooperativny multitasking iba **budia dojem** subezneho vykonavania uloh!

- Ako?
- Vykonavane ulohy (tasky) sa tak rychlo striedaju, ze ludske zmysly tuto vymenu nie su schopne postrehnut; naopak, cloveku sa javi beh jednej aplikacie spojito (napr. video prehravac)

Konkurentne programovanie

- Preemptivny multitasking sa tyka procesov a vlakien

Konkurentne programovanie

- Preemptivny multitasking sa tyka procesov a vlakien

- Kooperativny multitasking bol (velmi neuspesne) pouzity v OS Windows 3.x (95 a 98 pre beh 16-bitovych aplikacii)
- V sucasnosti zaziva velky rozmach nie na urovni OS, ale na aplikacnej urovni – async IO

Konkurentne programovanie

- !!! Async IO model programovania nie je zalozeny ani na vlaknach, ani na procesoch !!!

Konkurentne programovanie

- !!! Async IO model programovania nie je zalozeny ani na vlaknach, ani na procesoch !!!

- Na com je teda Async IO model zalozeny?

Konkurentne programovanie

- !!! Async IO model programovania nie je zalozeny ani na vlaknach, ani na procesoch !!!

- Na com je teda Async IO model zalozeny?
  - Procesu (vlaknu) OS prideluje CPU na isty cas

Konkurentne programovanie

- !!! Async IO model programovania nie je zalozeny ani na vlaknach, ani na procesoch !!!

- Na com je teda Async IO model zalozeny?
  - Procesu (vlaknu) OS prideluje CPU na isty cas
  - Ak beziaca uloha na CPU potrebuje cakat na vysledok IO operacie, OS prideli CPU inej ulohe

Konkurentne programovanie

- !!! Async IO model programovania nie je zalozeny ani na vlaknach, ani na procesoch !!!

- Na com je teda Async IO model zalozeny?
  - Procesu (vlaknu) OS prideluje CPU na isty cas
  - Ak beziaca uloha na CPU potrebuje cakat na vysledok IO operacie, OS prideli CPU inej ulohe
  - Preco vsak nevyuzit pridelene casove kvantum na vykonavanie inych casti programu?

Konkurentne programovanie

- Async IO paradigma (programovaci model) neprisiel na svet s jazykom Python

Konkurentne programovanie

- Async IO paradigma (programovaci model) neprisiel na svet s jazykom Python
- Oboznamujeme sa s nim sice pomocou jazyka Python, ale to vdaka tomu, ze UZ je podpora Async IO do jazyka Python integrovana

Konkurentne programovanie

- Async IO paradigma (programovaci model) neprisiel na svet s jazykom Python
- Oboznamujeme sa s nim sice pomocou jazyka Python, ale to vdaka tomu, ze UZ je podpora Async IO do jazyka Python integrovana

- Vid jazyky Go, C#, Scala, …

Konkurentne programovanie

- Paradigma Async IO nie je zalozena na vlaknach ani multiprocesingu, ale na jedno procesovom (jedno vlaknovom) modeli

Konkurentne programovanie

- Paradigma Async IO nie je zalozena na vlaknach ani multiprocesingu, ale na jedno procesovom (jedno vlaknovom) modeli

- Vyuziva kooperativny multitasking

Konkurentne programovanie

- Paradigma Async IO nie je zalozena na vlaknach ani multiprocesingu, ale na jedno procesovom (jedno vlaknovom) modeli

- Vyuziva kooperativny multitasking, cim **budi zdanie** subezneho vykonavania uloh

Konkurentne programovanie

- Na DU vid prehlad nastrojov konkurentneho programovania v Pythone (aj s vysvetlenim, kedy sa ktore hodi a na co je dobre)

- https://realpython.com/python-concurrency

Async IO

Async IO

- Ako by sme definovali pojem **asynchronny**?

Async IO

- Ako by sme definovali pojem **asynchronny**?
  1. Asynchronne rutiny su schopne prerusit svoj beh, pokym neobdrzia udaje na pokracovanie v behu. Tym umoznia inym rutinam vyuzit tento cas na ich beh.

Async IO

- Ako by sme definovali pojem **asynchronny**?
  1. Asynchronne rutiny su schopne prerusit svoj beh, pokym neobdrzia udaje na pokracovanie v behu. Tym umoznia inym rutinam vyuzit tento cas na ich beh.
  2. Asynchronny kod, v sulade s bodom 1, umoznuje konkurentne vykonavanie rutin (vyvolava zdanie ich subezneho vykonavania).

Async IO

- Ako je teda mozne dosiahnut konkurentne vykonavanie, ked mame k dispozicii jedno jadro CPU, jeden (jedno vlaknovy) proces?

- https://youtu.be/iG6fr81xHKA?t=4m29s

Async IO

- Sachova majsterka Judit Polgar sa predvadza a hra proti 24 superom. Ma dve moznosti ako hrat: synchronne alebo asynchronne

Async IO

- Sachova majsterka Judit Polgar sa predvadza a hra proti 24 superom. Ma dve moznosti ako hrat: synchronne alebo asynchronne
- Co vieme:
  - 24 protihracov
  - Judite trva tah 5 sekund
  - Amaterskemu hracovi trva tah 55 sekund
  - Jedna hra ma cca 30 tahov na kazdej strane (60 obaja hraci spolu)

Async IO

- Synchronna verzia hry

Async IO

- Synchronna verzia hry

- Judita hra jednu hru za druhou
- Jedna hra (55+5)*30 = 1800 sec = 30 min
- 24 hier: 24*30 = 720 min = 12 hod

Async IO

- Asynchronna verzia hry

Async IO

- Asynchronna verzia hry

- Judita sa medzi hracmi hybe (potiahne a ide k dalsiemu)
- Judita prejde vsetkych 24 stolov za 24*5=120 sec
- 120 > 55 sec, takze protihrac stihne urobit tah
- 1 hra 30 tahov: 120*30 = 3600 sec = 1 hod !!!

Async IO

- Async IO vyuziva casove intervaly, v ktorych nejaka funkcia (uloha) caka (je zablokovana) na beh tych funkcii (uloh), ktore zablokovane nie su.

Async IO

- Async IO vyuziva casove intervaly, v ktorych nejaka funkcia (uloha) caka (je zablokovana) na beh tych funkcii (uloh), ktore zablokovane nie su.

- Poznamka: funkcia, ktora je zablokovana cakanim na dokoncenie IO, znemoznuje vykonavanie akejkolvek inej casti kodu procesu (!!!jednovlaknovy proces!!!)

Async IO versus vlakna

Async IO versus vlakna

- V com je vyhoda Async IO oproti programovaciemu modelu zalozenemu na vlaknach?

Async IO versus vlakna

- V com je vyhoda Async IO oproti programovaciemu modelu zalozenemu na vlaknach?

- Vlakna zdielaju adresny priestor!!! Zdielaju spolocnu pamat procesu!!!

Async IO versus vlakna

- Pri programovani vlakien treba dbat na konkurentny pristup ku zdielanym udajovym strukturam

Async IO versus vlakna

- Pri programovani vlakien treba dbat na konkurentny pristup ku zdielanym udajovym strukturam

- Tejto problematike sme sa venovali v prvej casti semestra: aky synchronizacny mechanizmus sme pouzivali na ochranu integrity pamatoveho miesta pri sucasnom (modify/write) pristupe viacerych vlakien?

Async IO versus vlakna

- Tento problem pri Async IO odpada UPLNE a CELKOM

Async IO versus vlakna

- Tento problem pri Async IO odpada UPLNE a CELKOM

- Preco? Pretoze mame jednovlaknovu aplikaciu, takze v jednom case sa moze vykonavat iba jedna jedina instrukcia programu

Async IO versus vlakna

- To vsak neznamena, ze Async IO je lepsie riesenie nez vlakna

Async IO versus vlakna

- To vsak neznamena, ze Async IO je lepsie riesenie nez vlakna

- Ani to nie je pravda, ze tvorba aplikacie, ktora je plne Async IO, je jednoduche a lahke

Async IO versus vlakna

- To vsak neznamena, ze Async IO je lepsie riesenie nez vlakna

- Ani to nie je pravda, ze tvorba aplikacie, ktora je plne Async IO, je jednoduche a lahke
- Asyncio v Pythone je zalozene na entitach ako callbacks, events, transports, protocols, futures; znie to azda jednoducho?

async/await a generatory

async/await a generatory

- Python vo verzii 3.5 zaviedol nove klucove slova `async`/`await`, o ktorych este bude rec neskor

- Dovtedy nejestvovala v jazyku Python podpora tzv. nativnych korutin; korutiny boli implementovane pomocou rozsirenych generatorov

async/await a generatory

- Ako?

async/await a generatory

- Ako?

```
>>> def py34_coro_ver_a():
        for i in stuff():
            yield i


>>> def py34_coro_ver_b():
        yield from stuff()
```

async/await a generatory

- Podpora korutin zalozenych na generatoroch bude z jazyka Python odstranena vo verzii 3.10

- Presnejsie, tyka sa to dekoratora `@asyncio.coroutine`
- https://docs.python.org/3/library/asyncio-task.html#generator-based-coroutines

async/await a generatory

- Rozdiel medzi funkciou a generatorom

- Funkcia
  - Vsetko alebo nic
  - Ked sa raz zacne vykonavat, ukonci sa az prikazom return
- Generator
  - Prerusi svoj beh pri dosiahnuti riadku yield
  - Dokaze dodat hodnotu a obnovit svoj beh spolu so stavom lokalnych premennych

async/await a generatory

- Korutiny zalozene na generatoroch sme vyuzili na pochopenie problematiky
- V dalsom texte sa budeme venovat vylucne novej syntaxi jazyka, ktora podporuje nativne korutiny

async/await a nativne korutiny

async/await a nativne korutiny

- Zopakujme, co je korutina

async/await a nativne korutiny

- Zopakujme, co je korutina:

- Funkcia, ktora dokaze pozastavit svoje vykonavanie predtym, nez dosiahne koniec (pomocou prikazu return), a (nepriamo) dokaze na isty cas odovzdat vykonavanie inej korutine

async/await a nativne korutiny

- Skusme porovnat priklad synchronnej a asynchronnej verzie kratkeho ilustracneho programu

```python
#!/usr/bin/env python3
# countsync.py

import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```python
#!/usr/bin/env python3
# countsync.py

import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```
$ python3 countsync.py
One
Two
One
Two
One
Two
countsync.py executed in 3.01 seconds.
```

```python
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```python
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```
$ python3 countasync.py
One
One
One
Two
Two
Two
countasync.py executed in 1.01 seconds.
```

```python
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

async/await a nativne korutiny

- Synchronna verzia

async/await a nativne korutiny

- Synchronna verzia
  - Vykonava sekvencne 3x funkciu count()
  - Sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operacie

async/await a nativne korutiny

- Synchronna verzia
  - Vykonava sekvencne 3x funkciu count()
  - Sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operacie
- Asynchronna verzia

async/await a nativne korutiny

- Synchronna verzia
  - Vykonava sekvencne 3x funkciu count()
  - Sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operacie
- Asynchronna verzia
  - Sleep() vo funkcii count() nevykona zablokovanie funkcie, ale pozastavi jej vykonavanie, pokym vysledok (tu vyprsanie doby cakania) nebude k dispozicii a riadenie vrati “slucke udalosti”

async/await a nativne korutiny

- Synchronna verzia
  - Vykonava sekvencne 3x funkciu count()
  - Sleep() vo funkcii count() ilustruje dobu vykonania nejakej IO operacie
- Asynchronna verzia
  - Sleep() vo funkcii count() nevykona zablokovanie funkcie, ale pozastavi jej vykonavanie, pokym vysledok (tu vyprsanie doby cakania) nebude k dispozicii a riadenie vrati “slucke udalosti”
  - Slucka udalosti vybera na beh funkciu, ktora moze bezat

async/await a nativne korutiny

- Ked uloha pride k riadku “`await asyncio.sleep(1)`”, riadenie sa vrati slucke udalosti

async/await a nativne korutiny

- Ked uloha pride k riadku “`await asyncio.sleep(1)`”, riadenie sa vrati slucke udalosti
- “niekde” na pozadi sa poznamena, ze tuto ulohu je potrebne znovu “prebudit” o 1 sekundu

async/await a nativne korutiny

- Ked uloha pride k riadku “`await asyncio.sleep(1)`”, riadenie sa vrati slucke udalosti
- “niekde” na pozadi sa poznamena, ze tuto ulohu je potrebne znovu “prebudit” o 1 sekundu
- Medzitym sa moze vykonavat ina uloha

async/await a nativne korutiny

- Rozdiel verzii vidno na vystupoch

async/await a nativne korutiny

- Rozdiel verzii vidno na vystupoch:
  - Dlzka behu
  - Vypisy

async/await a nativne korutiny

- Rozdiel verzii vidno na vystupoch:
  - Dlzka behu
  - Vypisy

- Aj ked mame jedno vlakno, dosiahli sme zdanie sucasneho behu viacerych funkcii naraz!

async/await a nativne korutiny

- Syntax `async def` urcuje
  - Nativnu korutinu
  - Asynchronny generator

async/await a nativne korutiny

- Syntax `async def` urcuje
  - Nativnu korutinu
  - Asynchronny generator

- Vyrazy `async with` a `async for` su taktiez platne; vysvetlenie pride neskor

async/await a nativne korutiny

- Klucove slovo `await` odovzdava riadenie spat do cyklu spracovania udalosti (*event loop*, slucka udalosti)

- Dosledok: vykonavanie kodu sa prerusi

async/await a nativne korutiny

```
async def g():
    # Pause here and come back to g() when f() is ready
    r = await f()
    return r
```

- Vo funkcii `g()` nech je riadok `await f()`
- Riadenie sa vrati do cyklu spracovania udalosti spolu s informaciou, ze beh funkcie `g()` bude preruseny, pokym nebude k dispozicii vysledok volania funkcie `f()`
- Slucka udalosti moze medzitym nechat bezat inu ulohu (obnovit beh nejakej alebo nejaku spustit)

async/await a nativne korutiny

- Pravidla kedy a ako (ne)pouzivat async/await

async/await a nativne korutiny

- Pravidla kedy a ako (ne)pouzivat async/await

1. Definicia funkcie, pred ktorou sa nachadza `async`, je definiciou korutiny

2. Ako je (syntaktickou) chybou pouzit klucove slovo `yield` mimo `def` bloku, tak je chybou pouzit `await` mimo `async def` bloku

async/await a nativne korutiny

- Blok `async def`
  - Povolene `await`, `return`, `yield`
  - Vsetky tri volitelne
  - Platna definicia: `async def noop(): pass`

- Nie je mozne priamo zavolat takto definovanu korutinu, aby vratila nejaky vysledok!

async/await a nativne korutiny

```
>>> async def noop(): pass

>>> noop()
<coroutine object noop at 0x00000000038F90C0>
>>>
>>> await noop()
SyntaxError: 'await' outside function
>>>
```

async/await a nativne korutiny

```
>>> def run_noop():
        print(noop())


>>> run_noop()
<coroutine object noop at 0x00000000038F9140>

Warning (from warnings module):
  File "<pyshell#68>", line 2
RuntimeWarning: coroutine 'noop' was never awaited
>>>
```

async/await a nativne korutiny

```
>>> def run_noop():
        await noop()

SyntaxError: 'await' outside async function
>>>
```

async/await a nativne korutiny

```
>>> async def run_noop():
        await noop()


>>>
>>> run_noop()
<coroutine object run_noop at 0x00000000038F9140>
>>>
```

async/await a nativne korutiny

```
>>> import asyncio as aio
>>>
>>> aio.run(noop)
Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    aio.run(noop)
  File "C:\Program Files\Python38\lib\asyncio\runn
ers.py", line 37, in run
    raise ValueError("a coroutine was expected, go
t {!r}".format(main))
ValueError: a coroutine was expected, got <functio
n noop at 0x00000000038FA0D0>
>>>
```

async/await a nativne korutiny

```
>>>
>>> import asyncio as aio
>>>
>>> aio.run(noop())
>>>
```

async/await a nativne korutiny

- Ked piseme vo funkcii riadok `await f()`, znamena to, ze funkcia `f()` musi byt cakatelna (*awaitable*)

async/await a nativne korutiny

- Ked piseme vo funkcii riadok `await f()`, znamena to, ze funkcia `f()` musi byt cakatelna (*awaitable*)

- Bud ide tiez o korutinu (`async def xyz()`)
- Alebo o objekt, ktory implementuje metodu `__await()__`, ktora vracia iterator

async/await a nativne korutiny

- Ktore objekty su cakatelne?
- https://docs.python.org/3/reference/datamodel.html#awaitable-objects

async/await priklad

async/await priklad

- Majme program, ktory 3x vola ulohu `makerandom()`. Tato uloha generuje nahodne cisla v rozsahu <0;10>, pokym vygenerovane cislo nepresiahne hranicu urcenu argumentom ulohy.

async/await priklad

- Majme program, ktory 3x vola ulohu `makerandom()`. Tato uloha generuje nahodne cisla v rozsahu <0;10>, pokym vygenerovane cislo nepresiahne hranicu urcenu argumentom ulohy.
- Cielom je vytvorit taku verziu programu, aby mohli konkurentne bezat viacere instancie ulohy (teda nie seriovo za sebou)!

async/await priklad

```
import random
import time


def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")


def main():
    for i in range(3):
        makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    main()
```

```python
import random
import time


def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


def main():
    for i in range(3):
        makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    main()
    print(f"\nTime elapsed:{time.time()-t}")
```

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(0) == 10
Initiated makerandom(1).
makerandom(1) == 7 too low; retrying.
makerandom(1) == 8 too low; retrying.
makerandom(1) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(1) == 1 too low; retrying.
makerandom(1) == 6 too low; retrying.
---> Finished: makerandom(1) == 9
Initiated makerandom(2).
makerandom(2) == 3 too low; retrying.
---> Finished: makerandom(2) == 9

Time elapsed:23.11232590675354
>>>
```

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

async/await priklad

- Staci takato zmena, aby sa zo serioveho vykonavania stalo konkurentne pomocou asyncio?

async/await priklad

- Staci takato zmena, aby sa zo serioveho vykonavania stalo konkurentne pomocou asyncio?

- Pozrime si vystup!

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(0) == 10
Initiated makerandom(1).
makerandom(1) == 7 too low; retrying.
makerandom(1) == 8 too low; retrying.
makerandom(1) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(1) == 1 too low; retrying.
makerandom(1) == 6 too low; retrying.
---> Finished: makerandom(1) == 9
Initiated makerandom(2).
makerandom(2) == 3 too low; retrying.
---> Finished: makerandom(2) == 9

Time elapsed: 23.070329427719116
>>>
```

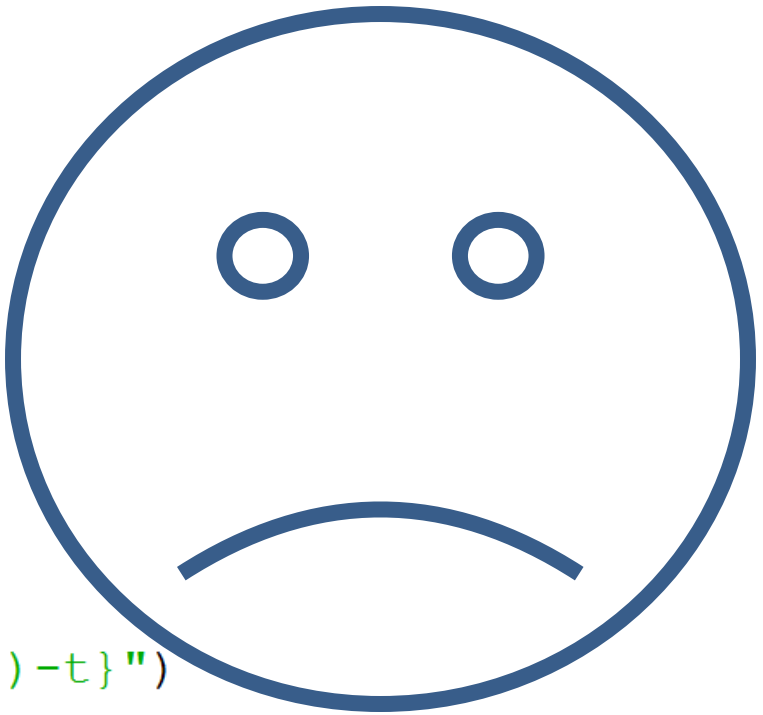
async/await priklad

- V com je problem, ked mame definovane vsetky potrebne nalezitosti asynchronnej aplikacie?

- Slucka udalosti (`asyncio.run()`)
- Asynchronne funkcie (`async`)
- Asynchronne volanie funkcii (`await`)

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```



async/await priklad

- Aj ked je funkcia `main`() definovana ako asynchronna a vyuziva asynchronne volanie dalsej korutiny, tak jednotlive volania korutiny su seriovo v cykle!

- Pomocou `asyncio.run`() sme spustili asynchronnu funkciu `main`(), ale korutiny spustame seriovo, nie konkurentne

async/await priklad

- V podstate mozeme vyuzit dve funkcie modulu `asyncio`, ktore implementuju konkurentne spustanie korutin, ktore su zadane ako argumenty

async/await priklad

- V podstate mozeme vyuzit dve funkcie modulu `asyncio`, ktore implementuju konkurentne spustanie korutin, ktore su zadane ako argumenty

- `asyncio.gather()` a `asyncio.wait()`
- https://stackoverflow.com/questions/42231161/asyncio-gather-vs-asyncio-wait

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    await asyncio.gather(*(makerandom(i, 10-i-1) for i in range(3)))


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(0) == 10
Initiated makerandom(1).
makerandom(1) == 7 too low; retrying.
makerandom(1) == 8 too low; retrying.
makerandom(1) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(1) == 1 too low; retrying.
makerandom(1) == 6 too low; retrying.
---> Finished: makerandom(1) == 9
Initiated makerandom(2).
makerandom(2) == 3 too low; retrying.
---> Finished: makerandom(2) == 9

Time elapsed:23.063318967819214
>>>
```

async/await priklad

- Blokujuce volanie vo funkcii `makerandom`!!!

async/await priklad

- Blokujuce volanie vo funkcii `makerandom`!!!

- Dosledky / priciny
  - Vlakno procesu musi cakat, zablokuje sa vykonavanie
  - Nemoze sa odovzdat riadenie slucke udalosti
  - Nikde vo funkcii `makerandom`() nie je `await`!

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        #time.sleep(idx+1)
        await asyncio.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    await asyncio.gather(*(makerandom(i, 10-i-1) for i in range(3)))


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

```
Initiated makerandom(0).
makerandom(0) == 4 too low; retrying.
Initiated makerandom(1).
makerandom(1) == 4 too low; retrying.
Initiated makerandom(2).
makerandom(2) == 0 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(1) == 7 too low; retrying.
makerandom(0) == 4 too low; retrying.
makerandom(2) == 4 too low; retrying.
makerandom(0) == 8 too low; retrying.
---> Finished: makerandom(1) == 10
makerandom(0) == 7 too low; retrying.
makerandom(0) == 8 too low; retrying.
makerandom(2) == 4 too low; retrying.
makerandom(0) == 7 too low; retrying.
makerandom(0) == 1 too low; retrying.
makerandom(0) == 6 too low; retrying.
---> Finished: makerandom(2) == 9
makerandom(0) == 3 too low; retrying.
makerandom(0) == 9 too low; retrying.
makerandom(0) == 7 too low; retrying.
---> Finished: makerandom(0) == 10

Time elapsed:12.163782835006714
>>>
```

async/await priklad

- Poznamky ku prikladu

- Priklad generovania nahodnych cisel nie je velmi vhodny pre `asyncio`…
- IO operacia je modelovana pomocou `sleep`!
- Rozne doby trvania IO operacie su modelovane pomocou argumentu `sleep`()

Async IO navrhove vzory

Async IO navrhove vzory

1. Zretazenie korutin

2. Producenti-konzumenti

Async IO navrhove vzory

- Zretazenie je (syntakticky) trivialne
- Netrivialny je logicky navrh programu ;)

- Vid ukazka chained.py

```python
async def part1(n: int) -> str:
    i = random.randint(0, 10)
    print(f"part1({n}) sleeping for {i} seconds.")
    await asyncio.sleep(i)
    result = f"--> res{n}-1"
    print(f"Returning part1({n}) == '{result}'.")
    return result

async def part2(n: int, arg: str) -> str:
    i = random.randint(0, 10)
    print(f"part2{n, arg} sleeping for {i} seconds.")
    await asyncio.sleep(i)
    result = f"{arg} --> res{n}-2"
    print(f"Returning part2{n, arg} == '{result}'.")
    return result

async def chain(n: int) -> None:
    p1 = await part1(n)
    p2 = await part2(n, p1)
    print(f"*** Chained res{n} => '{p2}'.")

async def main(*args):
    await asyncio.gather(*(chain(n) for n in args))

if __name__ == "__main__":
    random.seed(444)
    asyncio.run(main(1,2,3))
    print(f"Program finished.")
```

```
part1(1) sleeping for 4 seconds.
part1(2) sleeping for 4 seconds.
part1(3) sleeping for 0 seconds.
Returning part1(3) == '--> res3-1'.
part2(3, '--> res3-1') sleeping for 4 seconds.
Returning part1(1) == '--> res1-1'.
part2(1, '--> res1-1') sleeping for 7 seconds.
Returning part1(2) == '--> res2-1'.
part2(2, '--> res2-1') sleeping for 4 seconds.
Returning part2(3, '--> res3-1') == '--> res3-1 --> res3-2'.
*** Chained res3 => '--> res3-1 --> res3-2'.
Returning part2(2, '--> res2-1') == '--> res2-1 --> res2-2'.
*** Chained res2 => '--> res2-1 --> res2-2'.
Returning part2(1, '--> res1-1') == '--> res1-1 --> res1-2'.
*** Chained res1 => '--> res1-1 --> res1-2'.
Program finished.
>>>
```

Async IO navrhove vzory

- Na priklade vidime, ze `part2()` sa zacne vykonavat az po ukonceni korutiny `part1()`
- Toto spravanie sme tu uz dnes raz mali…

```python
import random
import time
import asyncio


async def makerandom(idx: int, threshold: int = 6) -> int:
    print(f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(f"makerandom({idx}) == {i} too low; retrying.")
        time.sleep(idx+1)
        i = random.randint(0, 10)
    print(f"---> Finished: makerandom({idx}) == {i}")
    return i


async def main():
    for i in range(3):
        await makerandom(i, 10-i-1)


if __name__ == "__main__":
    random.seed(444)
    t = time.time()
    asyncio.run(main())
    print(f"\nTime elapsed:{time.time()-t}")
```

Async IO navrhove vzory

- Na priklade vidime, ze `part2()` sa zacne vykonavat az po ukonceni korutiny `part1()`
- Toto spravanie sme tu uz dnes raz mali…
- V onom pripade to bolo neziaduce spravanie, pretoze sme chceli korutiny vykonavat konkurentne
- Niekedy vsak moze byt vykonavanie korutin od seba zavisle

```
part1(1) sleeping for 4 seconds.
part1(2) sleeping for 4 seconds.
part1(3) sleeping for 0 seconds.
Returning part1(3) == 'res3-1'.
part2(3, 'res3-1') sleeping for 4 seconds.
Returning part1(1) == 'res1-1'.
part2(1, 'res1-1') sleeping for 7 seconds.
Returning part1(2) == 'res2-1'.
part2(2, 'res2-1') sleeping for 4 seconds.
Returning part2(3, 'res3-1') == 'res3-2 from res3-1'.
-->Chained res3 => 'res3-2 from res3-1' (took 4.05 seconds).
Returning part2(2, 'res2-1') == 'res2-2 from res2-1'.
-->Chained res2 => 'res2-2 from res2-1' (took 8.06 seconds).
Returning part2(1, 'res1-1') == 'res1-2 from res1-1'.
-->Chained res1 => 'res1-2 from res1-1' (took 11.12 seconds).
Program finished in 11.12 seconds.
>>>
```

Async IO navrhove vzory

- Druhy vzor programovania Async IO uvedeny v tejto prednaske je model P-K

Async IO navrhove vzory

- Druhy vzor programovania Async IO uvedeny v tejto prednaske je model P-K

- PK problem vyzaduje “sklad”
- Python poskytuje modul `queue` (aj) na tieto ucely (“odolny” aj v pripade pouzitia vlakien)
- https://docs.python.org/3/library/queue.html#module-queue

Async IO navrhove vzory

- `asyncio` modul poskytuje podobnu funkcionalitu, urcenu pre asyncio pouzitie

- https://docs.python.org/3/library/asyncio-queue.html

Async IO navrhove vzory

- Kedy pouzit PK, kedy serializaciu mensich/kratsich korutin?

Async IO navrhove vzory

- Kedy pouzit PK, kedy serializaciu mensich/kratsich korutin?

- V pripade PK konzumenti nepoznaju ani pocet producentov, ani pocet poloziek, ktore treba spracovat
- P ani K navzajom priamo nekomunikuju, iba sprostredkovane cez polozky fronty

Async IO navrhove vzory

- Synchronna verzia riesenia PK je nepouzitelna

- Isty pocet producentov seriovo vkladaju polozky do fronty
- Az ked vsetci producenti ukoncia vkladanie, mozu zacat konzumenti vyberat polozky z fronty a spracuvat ich. Taktiez seriovo, polozku po polozke.

Async IO navrhove vzory

- Pri modeli PK moze byt problemom notifikacia konzumentov o tom, ze uz ziadne polozky do fronty nepribudnu
- Problemom je, aby konzumenti neostali navzdy cakat v metode q.get()

Async IO navrhove vzory

- Pri modeli PK moze byt problemom notifikacia konzumentov o tom, ze uz ziadne polozky do fronty nepribudnu

- Problemom je, aby konzumenti neostali navzdy cakat v metode q.get()

Async IO navrhove vzory

- Riesenie (hlavnej, koordinujucej ulohy):
  1. Vytvor frontu
  2. Vytvor ulohy producentov
  3. Vytvor ulohy konzumentov
  4. Pockaj na dokoncenie producentov
  5. Pockaj na vyprazdnenie fronty
  6. Zrus ulohy konzumentov

Async IO navrhove vzory

```
async def produce(n: int, q: asyncio.Queue) -> None:
    # Synchronous loop for each single producer
    for _ in range(n):
        item = await makeitem()
        await q.put(item)

async def consume(q: asyncio.Queue) -> None:
    while True:
        item = await q.get()
        await consume_item(item)
        q.task_done()

async def main(nprod: int, ncon: int) -> None:
    q = asyncio.Queue()
    producers = [asyncio.create_task(produce(n, q)) for n in range(nprod)]
    consumers = [asyncio.create_task(consume(n, q)) for n in range(ncon)]
    await asyncio.gather(*producers)
    await q.join()  # Implicitly awaits consumers, too
    for c in consumers:
        c.cancel()

if __name__ == "__main__":
    asyncio.run(main(3,10))
```

Cyklus spracovania udalosti

Cyklus spracovania udalosti

- Angl. *event loop*

- Predstavme si pre jednoduchost cyklus `while True`

Cyklus spracovania udalosti

- Angl. *event loop*

- Predstavme si pre jednoduchost cyklus `while True`, ktory
  - Monitoruje, ktore ulohy su necinne
  - Zistuje, ktora uloha moze byt vykonavana, kym ina sa stala necinnou (pozastavenou)
  - Obnovuje beh ulohy, pre ktoru je k dispozicii udalost, na ktoru cakala

Cyklus spracovania udalosti

- Cely tento menezment ohladom uloh sa da v Pythone (od 3.7) urobit jedinym riadkom!

```
asyncio.run(main())
```

Cyklus spracovania udalosti

- Cely tento menezment ohladom uloh sa da v Pythone (od 3.7) urobit jedinym riadkom!

```
asyncio.run(main())
```

- Tento riadok je zodpovedny za to, ze
  - Sa ziska instancia cyklu spracovania udalosti
  - Ulohy sa budu spustat, pokym nebudu oznacene ako ukoncene
  - Cyklus spracovania udalosti sa korektne ukonci

Cyklus spracovania udalosti

- Pokym nebola funkcia run() k dispozicii (alebo ak potrebujeme vacsiu kontrolu nad cyklom udalosti), vyuziva(l) sa nasledovny kod

Cyklus spracovania udalosti

- Pokym nebola funkcia run() k dispozicii (alebo ak potrebujeme vacsiu kontrolu nad cyklom udalosti), vyuziva(l) sa nasledovny kod

```
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

Cyklus spracovania udalosti

- Vzdy treba nejako zacat vykonavat program
- Pri asynchronnom programovani plati, ze vacsinou davame funkcii `run`() ako parameter hlavnu asynchronnu funkciu, ktora podla potreby vola dalsie

Cyklus spracovania udalosti

- Vzdy treba nejako zacat vykonavat program
- Pri asynchronnom programovani plati, ze vacsinou davame funkcii `run`() ako parameter hlavnu asynchronnu funkciu, ktora podla potreby vola dalsie

- V predoslej casti sme si uz ukazovali, ze asynchronnu funkciu nevieme spustit klasickym sposobom…

Konkurencia v Pythone

Konkurencia v Pythone

- Kedy zvolit *multiprocessing*, kedy *threading*, kedy *asyncio*?

- Mozu koexistovat v jednom programe?
- Ake kombinacie “su povolene”?

Konkurencia v Pythone

- Ked mame dobre skalovatelnu ulohu na nezavisle spracovanie na jednotlivych vypoctovych uzloch, a tieto uzly mame k dispozicii, jednoznacne **multiprocessing**

Konkurencia v Pythone

- Ked mame dobre skalovatelnu ulohu na nezavisle spracovanie na jednotlivych vypoctovych uzloch, a tieto uzly mame k dispozicii, jednoznacne **multiprocessing**
- Ak mame vyvijat aplikaciu, ktora vela casu travi spracovanim IO (sietova komunikacia, praca s externymi pamatami, komunikacia s pouzivatelom), jednoznacne **asyncio**

Konkurencia v Pythone

- Ked mame dobre skalovatelnu ulohu na nezavisle spracovanie na jednotlivych vypoctovych uzloch, a tieto uzly mame k dispozicii, jednoznacne **multiprocessing**
- Ak mame vyvijat aplikaciu, ktora vela casu travi spracovanim IO (sietova komunikacia, praca s externymi pamatami, komunikacia s pouzivatelom), jednoznacne **asyncio**
- Ostatne moze byt **threading**

Konkurencia v Pythone

- Treba mat na pamati, ze najtazsie sa ladia viacvlaknove aplikacie
- Zvlast preto, lebo casto nie je mozne zopakovat sled operacii tak, aby sa chyba dala zreprodukovat – velka miera nahodnosti

Konkurencia v Pythone

- Treba mat na pamati, ze najtazsie sa ladia viacvlaknove aplikacie
- Zvlast preto, lebo casto nie je mozne zopakovat sled operacii tak, aby sa chyba dala zreprodukovat – velka miera nahodnosti

- Avsak asynchronna aplikacia nie je zachranou a liekom na vsetko! V pripade uloh, ktore malo interaguju s IO, moze byt vysledna aplikacia nielen neprehladna, ale aj pomala

Konkurencia v Pythone

- Modul `asyncio` nie je jedinym, ktory v Pythone dokaze menezovat korutiny

- https://github.com/dabeaz/curio
- https://github.com/python-trio/trio
- …

Cvicenie

- Prepracovanie synchronnej aplikacie na asynchronnu
- Vyuzitie asynchronneho programovania pri spracovani HTTP ziadosti
- Porovnanie efektivity synchronneho versus asynchronneho pristupu

Zdroje

- Kratky, jasny a vystizny uvod do korutin: https://pymotw.com/3/asyncio/coroutines.html

- Vynikajuci prehlad vyvoja asynchronneho programovania v Pythone po 3.5: https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/

Zdroje

- Prednaska bola robena podla: https://realpython.com/async-io-python/

- Naco je dobre Async IO? Napriklad milion http requestov za 9 minut: https://pawelmhm.github.io/asyncio/python/aiohttp/2016/04/22/asyncio-aiohttp.html

Zdroje

- Pekne udrziavany dokladny a prehladny popis AsyncIO: https://yeray.dev/python/asyncio/asyncio-for-the-working-python-developer

PEP

- PEP 342 – Coroutines via Enhanced Generators
- PEP 380 – Syntax for Delegating to a Subgenerator
- PEP 3153 – Asynchronous IO support
- PEP 3156 – Asynchronous IO Support Rebooted: the “asyncio” Module
- PEP 492 – Coroutines with async and await syntax
- PEP 525 – Asynchronous Generators
- PEP 530 – Asynchronous Comprehensions