KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Úvod do programovacích stylů ◇ poznámky k přednášce

11. Asynchronní programování

verze z 4. prosince 2024

Korutina je funkce, u které je možné pozastavit vykonávání jejího těla. Generátory jsou tedy korutiny. Vezměme si například generátor:

```
def get_numbers():
    i = 0
    while i < 3:
        yield i
        i = i + 1
```

Nechme vygenerovat první prvek posloupnosti:

```
>>> c = get_numbers()
>>> next(c)
0
```

vykonávání těla se pozastavilo na řádku:

```
i = i + 1
```

Výraz `yield`:

```
(yield value)
```

je podobný příkazu `yield` až na to, že hodnotu výrazu musíme zadat z venčí. Říkáme, že korutině **zasíláme hodnotu**. Přesněji se výraz vyhodnotí tak, že vyprodukuje hodnotu *value* a pozastaví se vyhodnocování výrazu. Vyhodnocování pokračuje, až když se korutině zašle hodnota. **Přijatá hodnota** bude hodnotou `yield` výrazu.

Zasláním zprávy:

```
iterator.send(value)
```

zašleme korutině *iterator* hodnotu *value*.

Výraz:

```
next(iterator)
```

je ekvivalentní výrazu:

```
iterator.send(None)
```

Například:

```
def test(x):
    print("Start")
    val = yield x + 1
    print("Hodnota:", val)
```

Vytvoříme korutinu a spustíme jí:

```
>>> c = test(2)
>>> next(c)
Start
3
```

vyhodnocování výrazu:

```
yield x + 1
```

se pozastaví. Pokračuje, až když jí zašleme hodnotu:

```
>>> c.send(3)
Hodnota: 3
StopIteration
```

Korutina skončí.

Následující korutina po obdržení dvou čísel vrátí jejich součet:

```
def add():
    yield (yield) + (yield)
```

Vyzkoušíme:

```
>>> c = add()
>>> next(c)
>>> c.send(1)
>>> c.send(3)
4
>>> next(c)
StopIteration
```

1 Asynchronní korutiny

Speciální korutiny, kterým říkáme **asynchronní korutiny**, mohou pozastavit vykonávání do doby, než bude splněna zadaná podmínka. Asynchronní korutinu nazveme **spuštěnou**, pokud se začalo vykonávat její tělo. Při pozastavení asynchronní korutiny systém řeší, která korutina se splněnou podmínkou pro pokračování bude pokračovat ve vykonávání těla.

V této části budou všechny korutiny asynchronní. Knihovna `aio` (asynchronous input and output – asynchronní vstup a výstup) umožňuje práci s asynchronními korutinami. Knihovna se nalézá v souboru `aio.py` a importuje se příkazem:

```
import aio
```

Asynchronní korutinu je potřeba spustit funkcí:

```
aio.run(coroutine) => result
```

Volání funkce `run` skončí až po skončení korutiny *`coroutine`*. Hodnota *`result`* je návratová hodnota korutiny.

Funkce:

```
aio.sleep(delay) => coroutine
```

vrátí korutinu, která se ihned po spuštění pozastaví na *`delay`* vteřin a po obnovení vrátí `None`.

U asynchronních korutin upřednostníme zápis:

```
async aio.sleep(delay) => None
```

naznačující, že se jedná o korutinu. V zápise za šipkou uvádíme návratovou hodnotu korutiny.

Například:

```
>>> c1 = aio.sleep(1)
>>> aio.run(c1)
vteřina čekání
>>>
```

Funkce:

```
async aio.random_sleep(max_delay) => None
```

je podobná funkci `aio.sleep` až na to, že čeká náhodnou dobu mezi nula a *`max_delay`* vteřin.

Asynchronní korutinu vytvoříme přidáním `async` před příkaz `def` definující funkci. Například:

```
async def print_now(string):
    print(string)
```

Zkouška:

```
>>> c2 = print_now("A")
>>> aio.run(c2)
A
```

V těle asynchronní korutiny *`coroutine1`* můžeme použít výraz `await`:

```
(await coroutine2)
```

Výraz spustí korutinu *`coroutine2`* a pozastaví vykonávání korutiny *`coroutine1`* do doby, než *`coroutine2`* skončí. Návratová hodnota korutiny je pak hodnotou výrazu.

Například:

```
async def print_after(delay, string):
    await aio.sleep(delay)
    print(string)
```

Dostáváme:

```
>>> c3 = print_after(1, "A")
>>> aio.run(c3)
vteřina čekání
A
```

Funkce:

```
aio.arun(coroutine) => promise
```

spustí a ihned pozastaví korutinu *`coroutine`*. Funkce vrací **příslib** *`promise`*. Výraz `await` lze použít i na příslib. V takovém případě se vykonávání pozastaví do doby, než korutina *`coroutine`* skončí. Její návratová hodnota bude hodnotou `await` výrazu. Funkci `aio.arun` lze volat pouze během vykonávání funkce `aio.run`.

Například:

```
async def tasks():
    promise1 = aio.arun(print_after(2, "A"))
    promise2 = aio.arun(print_after(2, "B"))
    await promise1
    await promise2
    print("C")
```

Spustíme:

```
>>> aio.run(tasks())
dvě vteřiny čekání
A
B
C
```

2 Úložiště

Knihovna `aio` obsahuje jednoduché úložiště a simuluje komunikaci s ním přes síť. Úložiště vytvoříme instanciací třídy `Storage`:

```
>>> storage = aio.Storage()
```

Zpráva:

```
async storage.set(key, value) => storage
```

asynchronně uloží hodnotu *value* pod klíč *key* v úložišti *storage*.

Zpráva:

```
async storage.get(key) => value
```

asynchronně vrací hodnotu uloženou pod klíčem *key*. V případě neexistence klíče vyvolá chybu.

Úložiště má vlastnosti `max_delay` (maximální prodleva) a `error_probability` (pravděpodobnost chyby), které určují kvalitu simulovaného spojení s úložištěm přes síť. Maximální prodleva udává maximální dobu, po kterou může trvat přístup k serveru, ve vteřinách. Pravděpodobnost chyby je hodnota v intervalu $[0, 1]$ a udává pravděpodobnost, že při přístupu k úložišti nevydrží spojení. Výchozí maximální prodleva i pravděpodobnost chyby jsou nulové.

3 Asynchronní uživatelské rozhraní

Potřebujeme vytvořit asynchronní korutinu, která by cyklicky zpracovávala události uživatelského rozhraní. V knihovně `micro_widget` k tomuto účelu slouží procedura `mw.update_window`, která očekává jako argument hlavní okno. Procedura se stará o zpracování události všech otevřených oken.

Například program, který asynchronně v cyklu tiskne text pole:

```
w = mw.display_window()
e = mw.make_entry(w)

async def main():
    while mw.is_window_opened(w):
        mw.update_window(w)
        await aio.sleep(0.05)
        print(repr(mw.get_entry_text(e)))

aio.run(main())
```

Predikát `mw.is_window_opened` rozhoduje, zda je zadané okno otevřené.

V knihovně `omw` je potřeba k zpracování událostí uživatelského rozhraní zaslat zprávu `update` bez argumentů hlavnímu oknu.

Stejný program přepsaný do `omw` by vypadal následovně.

```
class EntryWindow(Window):
    def __init__(self):
        super().__init__()
        self.set_widget(Entry())

    def main_loop(self):
        aio.run(self.async_main_loop())

    async def async_main_loop(self):
        while self.is_opened():
            self.update()
            await aio.sleep(0.05)
            print(repr(self.get_widget().get_text()))

EntryWindow().main_loop()
```

Zpráva `is_opened` zaslaná oknu bez argumentů rozhodujeme, zda je okno otevřené.

Otázky a úkoly na cvičení

1. Vytvořte korutinu `memory` bez parametrů, která si uvnitř bude pamatovat jednu hodnotu. Po spuštění korutiny jí můžeme zasílat hodnoty. Korutina si přijatou hodnotu zapamatuje a vrátí předchozí zapamatovanou hodnotu. Výchozí zapamatovaná hodnota je `None`. Například:

   ```
   >>> c = memory()
   >>> next(c)
   >>> c.send(5)
   >>> c.send(4)
   5
   >>> c.send(3)
   4
   >>> c.send(10)
   3
   ```

2. Napište asynchronní korutinu `return_after`, která očekává dvě hodnoty ***`delay`*** a ***`value`***. Korutina po spuštění čeká ***`delay`*** vteřin a poté vrátí hodnotu ***`value`***.

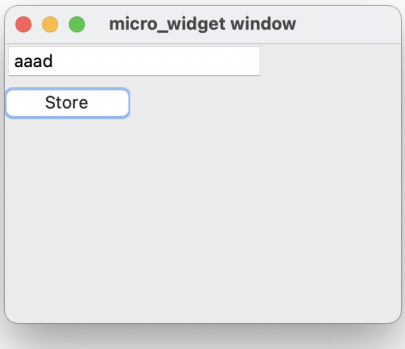
3. Napište asynchronní korutinu `corutine_add`, která očekává dvě nespuštěné asynchronní korutiny. Korutina `corutine_add` asynchronně spustí obdržené korutiny, čeká na jejich dokončení a vrátí součet návratových hodnot korutin. Například:

   ```
   >>> aio.run(corutine_add(return_after(2, 3), return_after(3, 4)))
   ### tři vteřiny čekání
   7
   ```

4. Napište asynchronní korutinu, která bude do nekonečna tisknou zadaný řetězec. Korutina mezi tisky bude čekat zadaný počet vteřin.

5. Tiskněte asynchronně každou vteřinu písmeno A a každé dvě vteřiny písmeno B. K tisku použijte funkci z předchozího úkolu.

6. Vytvořte okno s textovým polem a tlačítkem:



Zařiďte, aby po stisku tlačítka došlo k uložení zadaného textu do asynchronního úložiště. Dále v cyklu tiskněte obsah textu uloženého v úložišti. Tedy po uložení textu z příkladu se bude tisknout:

```
aaad
aaad
aaad
...
```

Vaše řešení musí fungovat i pro pomalé připojení k úložišti nastavené zasláním zprávy `set_max_delay`.

7. Upravte předchozí řešení tak, aby informovalo v okně uživatele o průběhu ukládání.

8. Upravte řešení předchozího úkolu tak, aby si poradilo i s chybami ve spojení nastavenými zasláním zprávy `set_error_probability` úložišti.

9. Vytvořte okno zobrazující aktuální čas získaný následovně.

```
import time

def get_current_time():
    return time.strftime("%H:%M:%S", time.gmtime())
```

10. Vytvořte okno s dvěma textovými poli a tlačítkem. Po stisku tlačítka se obsah polí uloží do asynchronního úložiště. Dále se zpět načtou uložená data a ty se zobrazí v polích v opačném pořadí. Efekt bude takový, že se po stisku tlačítka prohodí se zpožděním obsah polí.

11. Vylepšete řešení tak, aby fungovalo při pomalém a chybovém spojení s úložištěm. Uživatel musí být informován o tom, co se děje.

12. Vytvořte okno s textovým polem, které po každé změně uloží svůj obsah do úložiště.

13. Vytvořte jednoduchý úkolníček. Úkoly se budou ukládat do úložiště.