
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 29. listopadu 2023

Korutina je funkce, kde je možné pozastavit vykonávání 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 umožňuje získat hodnotu z vnějšku. Přesněji se výraz vyhodnotí tak, že vyprodukuje hodnotu *`value`* a pozastaví se vyhodnocování výrazu. Vyhodnocování pokračuje, až když program, který získal další hodnotu iterátoru, určí hodnotu výrazu.

Zasláním zprávy:

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

určíme, že aktuálně vyhodnocovaná hodnota výrazu `yield` v generátoru pro iterátor *`iterator`* bude *`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 dodáme hodnotu:

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

Korutina skončí.

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-output – asynchronní vstup-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

Pro práci s asynchronními korutinami vznikly třetí verze knihoven `micro_widget` a `omw`. Obě nové verze přidávají možnost spuštění korutiny zpracovávající uživatelské události okna. Tuto korutinu je potřeba spustit i v prostředí IDLE.

V knihovně `mw` funkce:

```
mw.main_loop(window, coroutine=None)
```

spustí korutinu zpracovávající události od uživatele. Argument *window* je hlavní okno. Pokud je zadaná korutina jako druhý argument, pak se spustí před zobrazením okna.

Podobně v knihovně `omw` zaslání zprávy:

```
window.main_loop(coroutine=None)
```

hlavnímu oknu spustí korutinu zpracovávající události od uživatele. Pokud je zadaná korutina jako argument, pak se spustí před zobrazením okna.