PROGRAMOWANIE WSPÓŁBIEŻNE (WIELOWĄTKOWE I WIELOPROCESOWE) PROGRAMOWANIE ASYNCHRONICZNE

Plan szkolenia

Plan modułu

TEORIA

1. **WĄTKI I PROCESY**
   - moduł *threading*
   - moduł *multiprocessing*

Wątki

- **Wątek** (*thread*) – przepływ sterowania w programie, który współdzieli stan globalny (pamięć) z innymi wątkami
- Wątki działające na pojedynczym procesorze/rdzeniu tworzą iluzję, że są wykonywane jednocześnie, chociaż zwykle działają z pewnym przeplotem
- Akcja jest nazywana **atomową**, jeśli gwarantuje się, że między początkiem, a końcem akcji nie nastąpi przełączanie wątków

Wątki w Pythonie

- Python oferuje wielowątkowość w dwóch wersjach:
  - starszy moduł *_thread* oferuje niskopoziomową funkcjonalność i nie jest zalecany do bezpośredniego użycia
  - nowszy moduł *threading* – jest modułem wyższego poziomu, zbudowanym na bazie modułu *_thread*
- Zaleca się użycie modułu *threading*
- Pełna ☞ dokumentacja modułu *threading*

Moduł *threading*

- Moduł *threading* definiuje kilka użytecznych funkcji:

Funkcje modułu *threading*

```
active_count()
current_thread()
enumerate()
stack_size([size])
```

FUNKCJA	DZIAŁANIE
active_count	liczba wątków aktywnych
current_thread	zwraca instancję bieżącego wątka
enumerate	lista wszystkich “żywych” wątków
stack_size	rozmiar stosu w bajtach wykorzystywany przez nowe wątki ustawienie *size* na 0 oznacza wybór wartości domyślnej podanie wartości nieakceptowanej przez system, powoduje zgłoszenie wyjątku *ValueError*

Uruchamianie wątków

- Instancje klasy *Thread* reprezentują wątki

Tworzenie instancji wątków

```
Thread(name=None, target=None, args=(), kwargs={})
```

- Zaleca się przekazywanie danych w formie argumentów nazwanych
- Zadanie realizowane przez wątek można podać na dwa sposoby:
  - jako funkcję przekazaną jako wartość parametru *target*
  - poprzez rozszerzenie klasy *Thread* i nadpisanie metody *run*
- Wątek może rozpocząć działanie dopiero po wywołaniu metody *start* na jego instancji

Uruchamianie wątków

Tworzenie i startowanie wątków – sposób 1

```
from threading import Thread, current_thread

def zadanie():
    nazwa_watka = current_thread().name
    for nr in range(1, 11):
        print(f'[{nazwa_watka}] hello #{nr:02d}')

w1 = Thread(target=zadanie, name='wątek1')
w2 = Thread(target=zadanie, name='wątek2')
w1.start()
w2.start()
```

Uruchamianie wątków

Tworzenie i startowanie wątków – sposób 2

```
from threading import Thread

class Watek(Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def run(self):
        nazwa_watka = self.name
        for nr in range(1, 11):
            print(f'[{nazwa_watka}] hello #{nr:02d}')


w1 = Watek(name='wątek1')
w2 = Watek(name='wątek2')
w1.start()
w2.start()
```

Uruchamianie wątków

- Nie można zakładać żadnej kolejności w jakiej wątki będą wykonywały swoje zadania – nie ma znaczenia kolejność startowania wątków
- Zawartość klasy *Thread*:

WŁAŚCIWOŚĆ	ZNACZENIE
daemon	właściwość określająca, czy wątek jest demoniczny (proces może się zakończyć, nawet, gdy wątki demoniczne są żywe – to kończy działanie wątków)
name	właściwość definiująca nazwę wątka

Uruchamianie wątków

- Klasa *Thread* definiuje także metody:

METODA	ZNACZENIE
is_alive	metoda testuje, czy wątek jest jeszcze “żywy” (został wystartowany, ale nie zakończył działania)
join	wstrzymuje działanie wątka wołającego, do momentu zakończenia działania wątka na którym metoda została wywołana
run	metoda definiująca zadanie wątka standardowo wywołuje metodę podaną przez parametr *target* nie należy metody wywoływać samodzielnie
start	powoduje, że wątek staje się aktywny i umożliwia wykonanie metody *run*

Synchronizacja wątków

- **Synchronizacja wątków** – mechanizm, który zapewnia, że dwa lub więcej współbieżnych wątków nie wykonuje jednocześnie określonego segmentu programu zwanego **sekcją krytyczną**
- Sekcja krytyczna odnosi się do części programu, w których uzyskuje się dostęp do **współdzielonego zasobu**
- Dostęp do sekcji krytycznej powinien odbywać się ze **wzajemnym wykluczeniem**

Synchronizacja wątków

- Jednoczesny dostęp do współdzielonych zasobów może prowadzić do zjawiska **wyścigu** (*race condition*)
- Ma to miejsce, gdy dwa lub więcej wątków może uzyskać dostęp do współdzielonych danych i próbują je zmienić w tym samym czasie
- W rezultacie wartości zmiennych mogą być nieprzewidywalne i różnić się w zależności od czasów przełączania kontekstu procesów

Synchronizacja wątków

Zjawisko wyścigu

```
from threading import Thread

class Licznik:
    def __init__(self):
        self.stan = 0

    def zwieksz(self):
        for _ in range(1_000_000):
            self.stan += 1

    def zmniejsz(self):
        for _ in range(1_000_000):
            self.stan -= 1

...
```

Synchronizacja wątków

Zjawisko wyścigu cd.

```
licznik = Licznik()
print(licznik.stan)  # 0

w1 = Thread(target=licznik.zwieksz)
w2 = Thread(target=licznik.zmniejsz)

w1.start()
w2.start()

# czekamy na zakończenie pracy wątków
w1.join()
w2.join()

print(licznik.stan)  # może być wartość inna niż 0
```

Blokady

- Moduł *threading* dostarcza kilku mechanizmów synchronizacji, umożliwiających wątkom komunikację i koordynację działania

Metody blokady *Lock*

```
acquire(blocking=True, timeout=-1)
release()
locked()
```

acquire	założenie i wejście w posiadanie blokady operacja może być blokująca lub nie
release	zwolnienie blokady metodę może wywołać wątek niebędący właścicielem blokady
locked	sprawdzenie, czy blokada jest założona

Blokady

- Obiekty blokady *RLock* posiadają identyczny zestaw metod jak blokada *Lock*
- Blokada *RLock* to **blokada wielowejściowa** (*re-entrant lock*), w której zaimplementowany jest mechanizm własności
- Tylko ten wątek, który założył blokadę może ją zwolnić
- Wątek posiadający blokadę może wielokrotnie wywołać metodę *acquire*, bez konieczności blokowania
- Blokada zostaje zwolniona, gdy metoda *release* zostanie wywołana tyle samo razy, co *acquire*
- Blokady implementują **protokół menedżera kontekstu**

Synchronizacja wątków

Użycie blokady do synchronizacji

```
from threading import Thread, Lock

class Licznik:
    def __init__(self):
        self.stan = 0

    def zwieksz(self, blokada):
        for _ in range(1_000_000):
            with blokada:          # blokada.acquire()
                self.stan += 1     # self.stan += 1
                                   # blokada.release()

    def zmniejsz(self, blokada):
        for _ in range(1_000_000):
            with blokada:
                self.stan -= 1

...
```

Synchronizacja wątków

Użycie blokady do synchronizacji cd.

```
licznik = Licznik()
print(licznik.stan)  # 0

blokada = Lock()

w1 = Thread(target=licznik.zwieksz, args=(blokada,))
w2 = Thread(target=licznik.zmniejsz, args=(blokada,))

w1.start()
w2.start()

# czekamy na zakończenie pracy wątków
w1.join()
w2.join()

print(licznik.stan)  # 0
```

Semafory

- **Semafory** (*semaphores*) to uogólnienie blokad
- Stanem blokady jest wartość logiczna (`True` lub `False`)
- Stanem semafora jest licznik (wartość pomiędzy 0, a podaną liczbą “przepustek”)
- Semafory mogą być użyteczne przy implementacji puli zasobów o określonym rozmiarze (można także do tego celu użyć kolejki *Queue*)
- Semafor reprezentują klasy *Semaphore* oraz *BoundedSemaphore*

Semafory

Metody semaforów

```
acquire(blocking=True)
release()
```

acquire	jeśli stan licznika jest dodatni, licznik jest dekrementowany i zwracana jest wartość `True` jeśli stan licznika ma wartość 0, a argument *blocking* – `True`, to wątek jest blokowany do momentu, aż inny wątek wywoła metodę *release* jeśli stan licznika ma wartość 0, a argument *blocking* – `False`, metoda od razu zwraca wartość `False`
release	jeśli stan licznika jest dodatni, lub licznik ma wartość 0 i nie ma żadnych wątków wstrzymanych, to licznik jest inkrementowany jeśli licznik semafora ma wartość 0 i są wątki oczekujące, to stan licznika nie ulega zmianie i jest wznawiany jeden z wątków wstrzymanych

Semafory

- Podczas tworzenia instancji semafora można zainicjować stan licznika (domyślnie licznik ma wartość 1)
- Jeśli licznik semafora *BoundedSemaphore* przekroczy wartość początkową, to zostanie zgłoszony wyjątek *ValueError*

Wielowątkowość w Pythonie – czy warto?

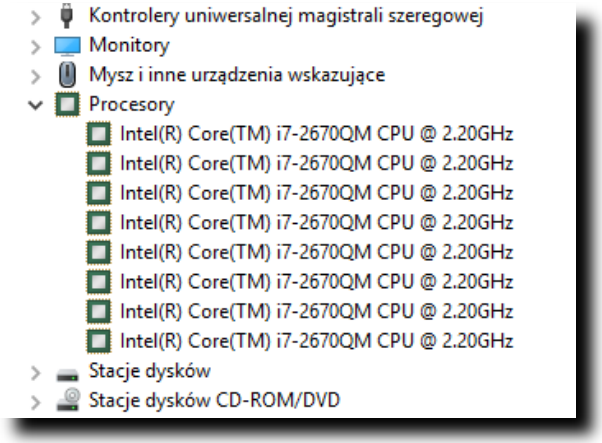
- Podstawowym pytaniem przed którym stajemy jest:
  Czy i kiedy warto wykorzystywać wielowątkowość?

- Spróbujmy przeprowadzić następujący eksperyment...
  - naszym zadaniem jest znalezienie wartości maksymalnej w dużej liście (kilka milionów elementów) zawierającej losowe wartości liczbowe
  - zadanie zostanie zrealizowane na dwa sposoby:
    - w “klasyczny sposób” wykorzystując iterację po całej liście
    - z użyciem dwóch wątków do których delegujemy zadanie znalezienia maksimum w połówce listy, następnie sprawdzimy, która wartość jest większa

Wielowątkowość w Pythonie – czy warto?

- W obu wariantach zmierzymy czas wykonania operacji (w drugim przypadku uwzględniamy także czas potrzebny na utworzenie wątków, ich zadań, wystartowanie, zaczekanie na wynik i wybór większej wartości)

- Aplikacja zostanie uruchomiona na maszynie 4-procesorowej (każdy procesor ma 2 rdzenie)
- Aplikacja działa na referencyjnej implementacji Pythona (*CPython*)
- Czego można się spodziewać?



Wielowątkowość w Pythonie – czy warto?

- Na pierwszy rzut oka wyniki mogą być zaskakujące. . .
- Wariant w którym zadanie zostało zdekomponowane na dwa niezależne podzadania jest znacząco wolniejszy
- Jest to spowodowane realizacją wielowątkowości w wybranych implementacjach Pythona, np. *CPython* (implementacja w C), *PyPy* (implementacja w Pythonie)
- Problem nie dotyczy takich implementacji Pythona jak *Jython* (implementacja w Javie), czy *IronPython* (implementacja w .NET)

Global Interpreter Lock

- “Winowajcą” jest tzw. **GIL** (*Global Interpreter Lock*)
- GIL jest muteksem, który pozwala tylko jednemu wątkowi na kontrolowanie interpretera Pythona
- W konsekwencji wątki mogą co prawda działać na różnych procesorach, ale w danej chwili będzie działać tylko jeden
- GIL ogranicza programowanie równoległe w Pythonie, nawet w architekturze wielowątkowej z więcej niż jednym rdzeniem procesora

Global Interpreter Lock

- Co ustaloną liczbę instrukcji kodu bajtowego GIL jest zwalniany, co pozwala działać wątkom pracującym poza interpreterem, czyli nieodwołującym się do API Pythona
- GIL jest zwalniany także w przypadku blokujących operacji wejścia/wyjścia, czyli np. przy odczycie czy zapisie do pliku
- Wątki w Pythonie nadają się do programów mocno obarczonych operacjami I/O
- Brak zysku przy operacjach czasochłonnych
- Potrzeba użycia rozwiązań opartych o system operacyjny

Procesy

- **Proces** to instancja uruchomionego programu
- Pakiet *multiprocessing* to pakiet, który umożliwia tworzenie i zarządzanie procesami przy użyciu interfejsu API podobnego do modułu *threading*
- Pakiet oferuje współbieżność lokalną i zdalną, przy użyciu podprocesów zamiast wątków
- W ten sposób można ominąć problemy związane z działaniem *Global Interpreter Lock* i w pełni wykorzystać wiele procesorów na danej maszynie
- Działa na systemach Unix i Windows
- Pełna ☞ dokumentacja pakietu *multiprocessing*

Procesy

- Procesy reprezentuje klasa *Process*
- Po utworzeniu instancji i przekazaniu zadania, proces należy wystartować (podobnie jak wątki)
- Należy zadbać o to, aby kod modułu głównego mógł być zaimportowany przez interpreter Pythona bez niezamierzonych efektów ubocznych (np. wystartowania nowego procesu)

Procesy

Tworzenie i startowanie procesów

```
from multiprocessing import Process, current_process
import os

def info():
    print('{:12} [id: {:5d}, rodzic: {:5d}]'.format(
        current_process().name,
        os.getpid(),
        os.getppid()))

if __name__ == "__main__":
    info()
    for i in range(4):
        Process(target=info).start()
```

```
MainProcess  [id:   824, rodzic:  8788]
Process-1    [id:  2856, rodzic:   824]
Process-2    [id:  2520, rodzic:   824]
Process-3    [id:  8880, rodzic:   824]
Process-4    [id:  8696, rodzic:   824]
```

Kolejki

- System operacyjny **izoluje** procesy między sobą
- Efektywne wykorzystanie wielu procesów zwykle wymaga pewnej **komunikacji** między nimi, tak aby można było równoważyć obciążenie i agregować wyniki
- Prostym sposobem komunikacji między procesami jest użycie **kolejki** do przekazywania komunikatów w obie strony
- Każdy obiekt serializowalny (z użyciem *pickle*) można przesłać przez kolejkę
- W ten sposób można uniknąć synchronizacji (np. z użyciem blokad)

Kolejki

- Kolejkę reprezentuje klasa *Queue*
- Zestaw metod jest podobny do metod klasy *Queue* z modułu *queue*
- Oprócz klasy *Queue* można także użyć klasy:

SimpleQueue	uproszczonej wersji kolejki
JoinableQueue	podklasy *Queue* oferującej dodatkowo metody: *task_done* oraz *join*

Kolejki

Przykład użycia kolejki

```
from multiprocessing import Process, Queue, current_process
import time

class Faks:
    def __init__(self, tresc):
        self.tresc = tresc

    def __str__(self):
        return self.tresc


def zadanie_serwera(kolejka):
    while True:
        dokument = kolejka.get()
        if not dokument:
            break
        time.sleep(1)
        print('Wysyłam faks', dokument)
```

Kolejki

Przykład użycia kolejki cd.

```
if __name__ == '__main__':
    kolejka_faksow = Queue()

    proces_serwera = Process(target=zadanie_serwera,
                             args=(kolejka_faksow,))
    proces_serwera.start()

    for i in range(3):
        faks = Faks('#{:d} od {}'.format(i,
                                         current_process().name))
        kolejka_faksow.put(faks)
    kolejka_faksow.put(None)

    kolejka_faksow.close()
    proces_serwera.join()
```

Blokady

- Innym sposobem synchronizacji procesów jest użycie blokad
- Moduł *multiprocessing* definiuje klasy będące odpowiednikami klas z modułu *threading*:
  - *BoundedSemaphore*
  - *Lock*
  - *RLock*
  - *Semaphore*

Pule

- W prawdziwym życiu trzeba uważać na tworzenie nieograniczonej liczby procesów roboczych
- Korzystanie z wielu procesów może przynieść korzyści w wydajności tylko wtedy, gdy liczba procesów jest równa lub bliska liczbie rdzeni w komputerze (p. metoda *cpu_count*)
- Wykonywanie większej liczby procesów roboczych niż ta optymalna wiąże się ze znacznymi dodatkowymi kosztami
- W konsekwencji typowym wzorcem projektowym jest **tworzenie puli z ograniczoną liczbą procesów roboczych** i przydzielanie im pracy
- Klasa *Pool* umożliwia realizację tego wzorca

Pule

Tworzenie instancji puli

```
Pool(processes=None, initializer=None, initargs=(),
     maxtasksperchild=None)
```

processes	liczba procesów w puli
initializer	opcjonalna funkcja wywoływana przy starcie każdego nowego procesu
initargs	argumenty przekazywane do funkcji inicjalizującej procesy
maxtasksperchild	maksymalna liczba zadań wykonywanych przez każdy proces puli

- Instancje puli oferują szereg metod
- Mogą być one wywołane tylko przez ten proces w którym została utworzona pula

Pule

Metody puli

```
apply(func, args=(), kwds={})
apply_async(func, args=(), kwds={}, callback=None)
```

apply	w dowolnym z procesów roboczych wywołuje funkcję z podanymi argumentami, w sposób synchroniczny i zwraca wynik
apply_async	w dowolnym z procesów roboczych wywołuje funkcję z podanymi argumentami, w sposób asynchroniczny i nie czekając na wynik zwraca instancję *AsyncResult* jeśli podano funkcję *callback*, to przekazywany jest jej wynik, gdy jest on gotowy funkcja nie powinna być czasochłonna, gdyż może zablokować proces

Pule

Metody puli

```
close()
imap(func, iterable, chunksize=1)
imap_unordered(func, iterable, chunksize=1)
```

close	nie można przesłać więcej zadań do puli procesy robocze kończą się, gdy zakończą wszystkie zaległe zadania
imap	zwraca iterator po wynikach wywołania podanej funkcji na kolejnych elementach obiektu iterowalnego *chunksize* określa, ile kolejnych elementów jest wysyłanych do każdego procesu
imap_unordered	podobnie do *imap*, ale kolejność nie jest ustalona

Pule

Metody puli

```
join()
map(func, iterable, chunksize=1)
map_async(func, iterable, chunksize=1, callback=None)
terminate()
```

join	czeka na zakończenie wszystkich procesów wcześniej należy wywołać *close* lub *terminate*
map	działa podobnie do *imap* ale zwraca listę wyników, a nie iterator
map_async	asynchroniczny wariant metody *imap*
terminate	kończy wszystkie procesy robocze natychmiast, bez czekania na zakończenie pracy

Pule

Przykład

```
from multiprocessing import Pool, current_process
import os
import time
import random

def jaka_dlugosc(tekst):
    time.sleep(random.random() * 2)   # symulacja czasochłonnej
                                      # operacji
    return len(tekst)

def zadanie(slowo):
    print('proces {:6}, {}'.format(os.getpid(),
                                   current_process().name))
    return jaka_dlugosc(slowo)

def dlugosci(tekst):
    with Pool() as pula:
        krotka = tuple(pula.imap(zadanie, tekst.split()))
    return krotka
```

Pule

Przykład cd.

```
if __name__ == '__main__':
    d = dlugosci('How I wish I could calculate pi')
    print('\nwynik: ', *d, sep='')
```

```
proces   2244, SpawnPoolWorker-1
proces  12416, SpawnPoolWorker-2
proces   7960, SpawnPoolWorker-4
proces  11588, SpawnPoolWorker-3
proces  12360, SpawnPoolWorker-5
proces   6664, SpawnPoolWorker-6
proces   6488, SpawnPoolWorker-7

wynik: 3141592
```

Procesy demoniczne

- Procesy, podobnie jak wątki, posiadają flagę *daemon*
- Wartość tej flagi (`True` lub `False`) można ustawić zanim proces zostanie wystartowany (metoda *start*)
- Kiedy zwykły (niedemoniczny) proces kończy pracę, próbuje zakończyć wszystkie swoje demoniczne procesy potomne – procesy demoniczne nie mają wpływu na całkowity czas działania aplikacji
- Proces demoniczny nie może tworzyć procesów potomnych
- W przeciwnym razie mógłby pozostawić swoje procesy potomne “osierocone”, gdyby sam został zakończony po zakończeniu swojego procesu-rodzica

Wymiana danych pomiędzy procesami

- Zazwyczaj najlepiej jest unikać udostępniania stanu między procesami – zamiast tego można użyć kolejek do jawnego przekazywania między nimi komunikatów
- Jednak w sytuacjach, w których trzeba współdzielić stan, moduł *multiprocessing* dostarcza klas dostępu do **pamięci współużytkowanej** (*shared memory*)

Wymiana danych pomiędzy procesami

Pamięć współdzielona

```
Value(typecode, *args, lock=True)
Array(typecode, size_or_initializer, lock=True)
```

Value	klasa do przechowywania pojedynczej wartości wspólnej dla dwóch lub więcej procesów
Array	klasa do przechowywania ustalonej ilości wartości prostych (tego samego typu)

- Blokadę można uzyskać poprzez wywołanie metody *get_lock*
- Do odczytu/ustawienia wartości służy atrybut *value*

Wymiana danych pomiędzy procesami

- Bardziej elastycznym rozwiązaniem, umożliwiającym m.in. koordynację między różnymi komputerami w sieci (nie współdzielącymi pamięci) jest użycie klasy *Manager*
- Jest ona podklasą klasy *Process* (z tymi samymi metodami i atrybutami)
- Instancja klasy steruje procesem serwera, który zarządza obiektami współużytkowanymi
- Inne procesy mogą uzyskiwać dostęp do udostępnionych obiektów za pośrednictwem obiektów proxy $\rightarrow$ większy narzut

Wymiana danych pomiędzy procesami

Przykład

```
from multiprocessing import Manager, Process, current_process
import os
import time
import random

def jaka_dlugosc(s):
    time.sleep(random.random() * 2)  # symulacja czasochłonnej operacji
    return len(s)

def zadanie(nr, slowo, slownik):
    print('proces {:6}, {}'.format(os.getpid(), current_process().name))
    slownik[nr] = jaka_dlugosc(slowo)

def dlugosci(tekst):
    mgr = Manager()
    slownik = mgr.dict()
    procesy = []
    for nr, s in enumerate(tekst.split()):
        p = Process(target=zadanie, args=(nr, s, slownik))
        p.start()
        procesy.append(p)
    for p in procesy:
        p.join()
    return [dlugosc for _, dlugosc in sorted(slownik.items())]
```

Wymiana danych pomiędzy procesami

Przykład cd.

```
if __name__ == '__main__':
    d = dlugosci('How I wish I could calculate pi')
    print('\nwynik: ', *d, sep='')

proces  12404, Process-2
proces  11368, Process-3
proces   9472, Process-4
proces   7748, Process-5
proces   8952, Process-6
proces  10104, Process-7
proces   8180, Process-8

wynik: 3141592
```

Plan szkolenia

Plan modułu

TEORIA

2 ASYNCHRONICZNY PYTHON

- moduł ASYNCIO – podstawowe zagadnienia

Biblioteka asyncio

- Biblioteka do programowania asynchroniczego
  - opiera się na coroutines (współprogram)
  - “funkcje” współpracujące, które dobrowolnie oddają kontrolę
  - możliwe jest zawieszenie wykonywania programu i przeniesienie wykonywania do innego współprogramu
  - w przeciwieństwie do subroutines (podprogram, funkcja) które muszą być wywłaszczone
  - wykonywanie naprzemiennie (*concurrent*), ale nie równolegle (*parallel*)

Coroutines w Pythonie

- Zaimplementowane na bazie generatorów
- Dobrowolnie oddają wykonywanie
- W momencie oddania kontroli zapamiętują swój stan (patrz *yield*)
- Mogą kontynuować swój program od tego momentu (patrz *next* oraz *send*)
- Zminimalizowany problem wyścigu (*race conditions*)

async def, await, event_loop

- asyncio dostarcza nam *event_loop*, który zarządza wykonywaniem kodu asynchronicznego (coroutynami)
- Słowo kluczowe **`async def`** definiuje nam coroutynę (jako funkcję):
  async def coroutine(args): ...
- Wyrażenie *await* w bloku coroutyny może wstrzymać jej wykonywanie dopóki wartość argumentu jest gotowa:
  val = await awaitable

Awaitable

- *Awaitable* to podstawowy typ w *asyncio* na którego wynik można oczekiwać (w sposób asynchroniczny)
- *Coroutine*, *Future* oraz *Task* to podklasy *awaitable*
- Jest ona “awaitable" – można na niej wykonać:
  val = await awaitable
- Klasa definująca metodę *__await__*

Future

- Reprezentuje jednostkę, która jest w trakcie wykonywania
- Wynik jej działania może nie być dostępny natychmiast
- Oczekiwanie na *future* oddaje wykonywanie do *event_loop* dopóki wynik jest gotowy
- Posiadają również synchroniczny interfejs:
  - f.done()
  - f.result()
  - f.exception()

Task

- Podstawowa jednostka wykonywana przez *event_loop*
- *event_loop* może w danym momencie wykonywać tylko jeden task, pozostałe taski w tym momencie oczekują
- Jest podklasą *Future*
- Jest wykorzystywana do wykonywania corutyn
- Jest gotowy kiedy corutyna zakończyła działanie

Dodatkowe konstrukcje

- *asyncio.run(future)* powoduje wykonanie *future* (w tym tasku) w domyślnym *event_loop* do momentu, aż wynik *future* jest dostępny
  - w przypadku coroutine zostanie ona opakowana w task
- *await asyncio.sleep(0)* oddaje wykonywanie
- Dodatkowe wyrażenia:
  - *async for* dla (async) iteratorów, które zwracają *awaitables*
  - *async with* dla (async) contextmanagerów, które oczekują na zajęcie lub zwolnienie zasobu

sync in async

- Nie powinniśmy wykonywać synchronicznych operacji blokujących w kodzie asynchronicznym
- Kod blokujący wykonujemy za pomocą: *await loop.run_in_executor(blocking_func)*
- Zostanie ona wykonana w *threadpool* albo *processpool*

Dziękujemy za uwagę