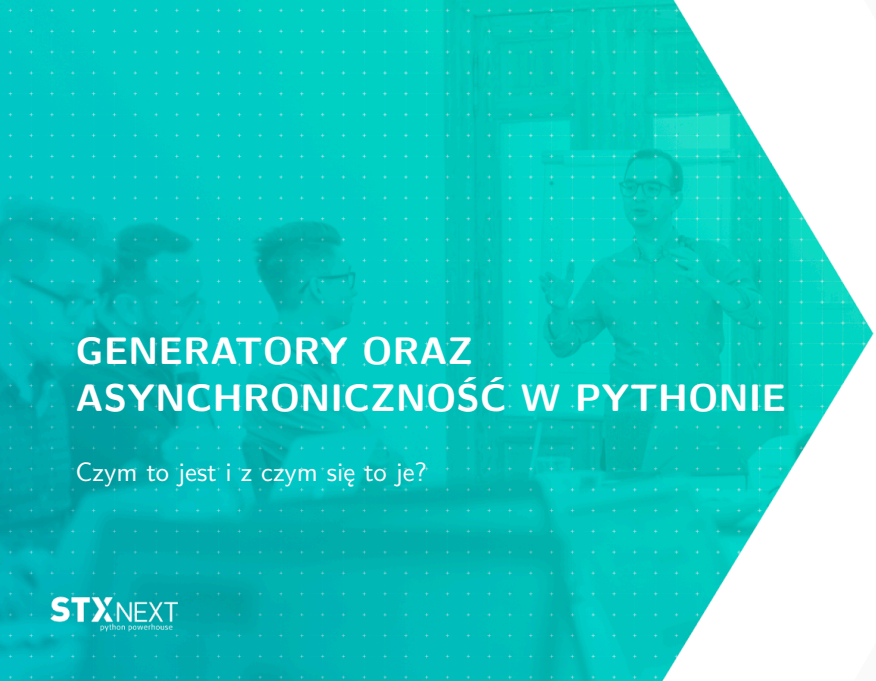


GENERATORY ORAZ ASYNCHRONICZNOŚĆ W PYTHONIE

Czym to jest i z czym się to je?

STX NEXT
python powerhouse

Przykładowy problem

Załóżmy, że dostaliśmy zadanie, w którym musimy odczytać dużą ilość danych, przetworzyć ją w jakiś sposób i kazać użytkownikowi podjąć jakąś decyzję co do ich wartości.

```
1  def main():
2      data = getAllData()
3
4      processed_data = []
5      for rawData in data:
6          processed_data.append(processData(rawData))
7
8      result = []
9      for datapoint in processed_data:
10         if askUserForAction(datapoint):
11             result.append(datapoint)
12
13     saveData(result)
```

Przykładowy problem

Załóżmy, że dostaliśmy zadanie, w którym musimy odczytać dużą ilość danych, przetworzyć ją w jakiś sposób i kazać użytkownikowi podjąć jakąś decyzję co do ich wartości.

Jednak po oddaniu programu, nasz użytkownik zwrócił nam uwagę że program ładuje się bardzo długo...

```
1  def main():
2      data = getAllData()
3
4      processed_data = []
5      for rawData in data:
6          processed_data.append(processData(rawData))
7
8      result = []
9      for datapoint in processed_data:
10         if askUserForAction(datapoint):
11             result.append(datapoint)
12
13     saveData(result)
```

Jak możemy ten program przyspieszyć?

```
1  def main():
2      data = getAllData()
3
4      processed_data = []
5      for rawData in data:
6          processed_data.append(processData(rawData))
7
8      result = []
9      for datapoint in processed_data:
10         if askUserForAction(datapoint):
11             result.append(datapoint)
12
13     saveData(result)
```

Pierwsze podejście

Spróbujmy pobierać dane po jednej wartości na raz.

```
1  def main():
2      result = []
3      while True:
4          data = getOneData()
5          if data is None:
6              # no data left to process
7              break
8
9          processed_data = processData(data)
10
11         if askUserForAction(processed_data):
12             result.append(datapoint)
13
14     saveData(result)
```

Pierwsze podejście

Spróbujmy pobierać dane po jednej wartości na raz.

Aplikacja działa, ale pojawia się nam jakiś dziwny `if` z warunkiem wyjścia z pętli

```
1  def main():
2      result = []
3      while True:
4          data = getOneData()
5          if data is None:
6              # no data left to process
7              break
8
9          processed_data = processData(data)
10
11         if askUserForAction(processed_data):
12             result.append(datapoint)
13
14     saveData(result)
```

Pierwsze podejście

Spróbujmy pobierać dane po jednej wartości na raz.

Aplikacja działa, ale pojawia się nam jakiś dziwny `if` z warunkiem wyjścia z pętli

Czy da się ten kod napisać ładniej?

```
1  def main():
2      result = []
3      while True:
4          data = getOneData()
5          if data is None:
6              # no data left to process
7              break
8
9          processed_data = processData(data)
10
11         if askUserForAction(processed_data):
12             result.append(datapoint)
13
14     saveData(result)
```

Czym jest generator?

Z dokumentacji `pythona` dowiemy się że:

Czym jest generator?

Z dokumentacji `pythona` dowiemy się że:

Dokumentacja

[Generator is] a function which returns a **generator iterator**. It looks like a **normal function** except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Czym jest generator?

Z dokumentacji `pythona` dowiemy się że:

Dokumentacja

[Generator is] a function which returns a **generator iterator**. It looks like a **normal function** except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Co to oznacza po polsku?

Czym funkcjo-podobnym czymś jest generator?


```
1  def generator():
2    # this will be a generator function
```

Czym funkcjo-podobnym czymś jest generator?


```
1  def generator():
2    # this will be a generator function
3    yield "foo"
```

Czym funkcjo-podobnym czymś jest generator?


```
1  def generator():
2    # this will be a generator function
3    yield "foo"
4    yield "bar"
```

Czym funkcjo-podobnym czymś jest generator?

```
1  def generator():
2    # this will be a generator function
3    yield "foo"
4    yield "bar"
5
6
7  print(generator())
```

Spróbujmy uruchomić naszą „funkcje”

Czym funkcjo-podobnym czymś jest generator?

```
1  def generator():
2    # this will be a generator function
3    yield "foo"
4    yield "bar"
5
6
7  print(generator())
   <generator object generator at 0x7fa85e944200>
```

Spróbujmy uruchomić naszą „funkcje”

Zamiast naszego tekstu dostaliśmy jakiś dziwny obiekt...

Powrót do dokumentacji

Definicja mówiła coś o funkcji `next()`, spróbujmy jej użyć.

```
1  def generator():
2      # this will be a generator function
3      yield "foo"
4      yield "bar"
5
6
7  wierd_obj = generator()
8  print(next(wierd_obj))
```

Powrót do dokumentacji

Definicja mówiła coś o funkcji `next()`, spróbujmy jej użyć.

Otrzymaliśmy naszą wartość `foo`, spróbujmy jeszcze raz użyć `next()`.

```
1  def generator():
2      # this will be a generator function
3      yield "foo"
4      yield "bar"
5
6
7  wierd_obj = generator()
8  print(next(wierd_obj))
   foo
```

Powrót do dokumentacji

Definicja mówiła coś o funkcji `next()`, spróbujmy jej użyć.

Otrzymaliśmy naszą wartość `foo`, spróbujmy jeszcze raz użyć `next()`.

Pojawiło się nasze `bar`, co się stanie jeśli jeszcze raz użyjemy `next()`?

```
1  def generator():
2      # this will be a generator function
3      yield "foo"
4      yield "bar"
5
6
7  wierd_obj = generator()
8  print(next(wierd_obj))
   foo
9  print(next(wierd_obj))
   bar
```

Powrót do dokumentacji

Definicja mówiła coś o funkcji `next()`, spróbujmy jej użyć.

Otrzymaliśmy naszą wartość `foo`, spróbujmy jeszcze raz użyć `next()`.

Pojawiło się nasze `bar`, co się stanie jeśli jeszcze raz użyjemy `next()`?

```
1  def generator():
2      # this will be a generator function
3      yield "foo"
4      yield "bar"
5
6
7  wierd_obj = generator()
8  print(next(wierd_obj))
   foo
9  print(next(wierd_obj))
   bar
10 print(next(wierd_obj))
   Traceback (most recent call last):
     File "simpleGen.py", line 10, in <module>
       print(next(wierd_obj))
   StopIteration
```

Generator jako iterator

Nasza *„funkcja”* zwraca nam jakiś iterator, który zwraca po kolei wszystkie wartości z wyrażeń `yield`, a po zakończeniu wyrzuca jakieś `Exception`. Dodajmy parę `print`ów by zobaczyć co się dzieje.

```
1  def generator():
2    print("here")
3    yield "foo"
4    print("there")
5    yield "bar"
6    print("done")
7
8
9  gen = generator()
10 print(next(gen))
```

Generator jako iterator

Nasza *„funkcja”* zwraca nam jakiś iterator, który zwraca po kolei wszystkie wartości z wyrażeń `yield`, a po zakończeniu wyrzuca jakieś `Exception`. Dodajmy parę `print`ów by zobaczyć co się dzieje.

```
1  def generator():
2    print("here")
3    yield "foo"
4    print("there")
5    yield "bar"
6    print("done")
7
8
9  gen = generator()
10 print(next(gen))
   here
   foo
```

Generator jako iterator

Nasza *„funkcja”* zwraca nam jakiś iterator, który zwraca po kolei wszystkie wartości z wyrażeń `yield`, a po zakończeniu wyrzuca jakieś `Exception`. Dodajmy parę `print`ów by zobaczyć co się dzieje.

```
1  def generator():
2    print("here")
3    yield "foo"
4    print("there")
5    yield "bar"
6    print("done")
7
8
9  gen = generator()
10 print(next(gen))
   here
   foo
11 print(next(gen))
   there
   bar
```

Generator jako iterator

Nasza *„funkcja”* zwraca nam jakiś iterator, który zwraca po kolei wszystkie wartości z wyrażeń `yield`, a po zakończeniu wyrzuca jakieś `Exception`. Dodajmy parę `print`ów by zobaczyć co się dzieje.

```
1  def generator():
2    print("here")
3    yield "foo"
4    print("there")
5    yield "bar"
6    print("done")
7
8
9  gen = generator()
10 print(next(gen))
   here
   foo
11 print(next(gen))
   there
   bar
12 print(next(gen))
   done
   Traceback (most recent call last):
   [...]
```

To czym jest generator?

- Nasza *„funkcja”* `generator` jest tak naprawdę konstruktorem.

To czym jest generator?

- Nasza *„funkcja”* `generator` jest tak naprawdę konstruktorem.
- Nasz kod jest wykonywany dopiero wtedy, kiedy poprosimy o nową wartość.

To czym jest generator?

- Nasza *„funkcja”* `generator` jest tak naprawdę konstruktorem.
- Nasz kod jest wykonywany dopiero wtedy, kiedy poprosimy o nową wartość.
- Czy możemy wykożystać naszą funkcje w pętli `for in`?

Generator oraz pętle `for in`

Skorzystajmy zatem z naszej *„funkcji”* w pętli:

```
1  def generator():
2      yield 1
3      yield 2
4      yield 3
5
6
7  for n in generator():
8      print(n)
```

Generator oraz pętle `for in`

Skorzystajmy zatem z naszej *„funkcji”* w pętli:

```
1  def generator():
2      yield 1
3      yield 2
4      yield 3
5
6
7  for n in generator():
8      print(n)
```

```
1
2
3
```

A co jeśli stworzymy nieskończony generator?


```
1  def silnia():
2      yield 1
3      yield 1
4      krok = 3
5      wynik = 2
6      while True:
7          yield wynik
8          wynik = wynik * krok
9          krok += 1
10
11
12 for n in silnia():
13     print(n)
```

A co jeśli stworzymy nieskończony generator?

```
1  def silnia():
2      yield 1
3      yield 1
4      krok = 3
5      wynik = 2
6      while True:
7          yield wynik
8          wynik = wynik * krok
9          krok += 1
10
11
12 for n in silnia():
13     print(n)
```

```
1
1
2
6
24
120
720
5040
40320
362880
[...]
```

Jak możemy poprawić nasz kod z użyciem generatorów?

Możemy napisać generator zwracający przetworzone wartości, jedna po drugiej.

```
1  def getDataGenerator():
2      while True:
3          data = getOneData()
4          if data is None:
5              # no data left to process
6              break
7          # return data from generator
8          yield processData(data)
9
10
11 def main():
12     result = []
13
14     for datapoint in getDataGenerator():
15         if askUserForAction(datapoint):
16             result.append(datapoint)
17
18     saveData(result)
```

Jak możemy poprawić nasz kod z użyciem generatorów?

Możemy również przepisać naszą pętle która prosi użytkownika o podjęcie decyzji na generator.

```
1  def getDataGenerator():
2      while True:
3          data = getOneData()
4          if data is None:
5              # no data left to process
6              break
7          yield processData(data)
8
9
10 def gatherUserActions(data):
11     for datapoint in data:
12         if askUserForAction(datapoint):
13             yield datapoint
14
15
16 def main():
17     saveData(
18         gatherUserActions(getDataGenerator())
19     )
```

Podsumowanie

Generatory pozwalają nam pisać kod który generuje kolejne wartości dopiero wtedy kiedy są potrzebne. Robi to przez przerywanie wykonywania programu i zwracanie wartości za pomocą słowa kluczowego `yield`.

Nowy problem

Program jest nadal wolny, ale inaczej...

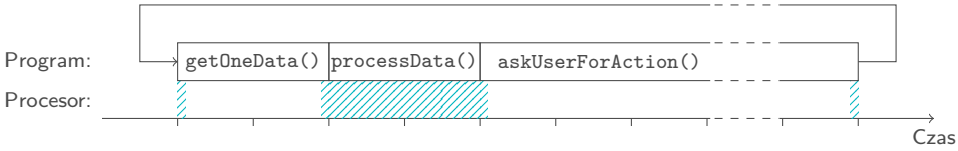
Po oddaniu naszego programu przepisanego na nowo poznane generatory, doszły do nas słuchy, że program może szybciej się ładuje na początku, ale użytkownicy muszą dłużej czekać na załadowanie się nowego przypadku.

Nowy problem

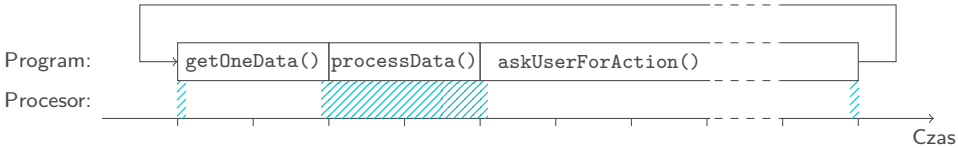
Program jest nadal wolny, ale inaczej...

Po oddaniu naszego programu przepisanego na nowo poznane generatory, doszły do nas słuchy, że program może szybciej się ładuje na początku, ale użytkownicy muszą dłużej czekać na załadowanie się nowego przypadku.
Użycie superkomputera nie pomogło.

Analiza problemu



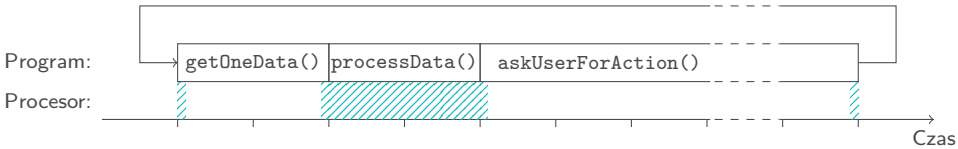
Analiza problemu



Wnioski:

- Nasz program spędza dużą część swojego czasu na czekaniu.

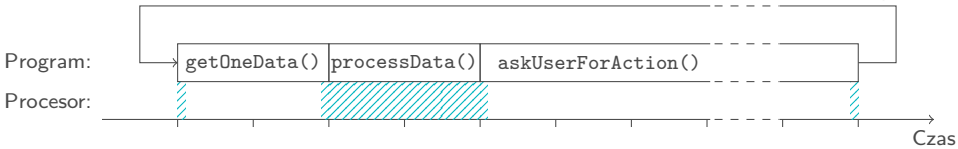
Analiza problemu



Wnioski:

- Nasz program spędza dużą część swojego czasu na czekaniu.
- Cały okres oczekiwania, który kiedyś był na początku programu, został rozbity na mniejsze między akcjami dla użytkownika.

Analiza problemu



Wnioski:

- Nasz program spędza dużą część swojego czasu na czekaniu.
- Cały okres oczekiwania, który kiedyś był na początku programu, został rozbity na mniejsze między akcjami dla użytkownika.
- Czy możemy pobierać, przetwarzać, i pytać użytkownika o decyzje w tym samym czasie?

Na co nasz program czeka?

Za każdym razem, kiedy nasz program chce czytać dane nie znajdujące się w jego pamięci, to musi prosić system operacyjny o ich odczytanie. W tym czasie wykonywanie naszego programu jest zablokowane.

Na co nasz program czeka?

Za każdym razem, kiedy nasz program chce czytać dane nie znajdujące się w jego pamięci, to musi prosić system operacyjny o ich odczytanie. W tym czasie wykonywanie naszego programu jest zablokowane.

Nazywamy taki dostęp do danych *synchronicznym*.

Na co nasz program czeka?

Za każdym razem, kiedy nasz program chce czytać dane nie znajdujące się w jego pamięci, to musi prosić system operacyjny o ich odczytanie. W tym czasie wykonywanie naszego programu jest zablokowane.

Nazywamy taki dostęp do danych *synchronicznym*.

Czy możemy uzyskiwać dostęp do naszych danych inaczej?

Asynchroniczne I/O

Gdy zapytamy wikipedię, czym jest asynchroniczne I/O, to otrzymamy taką definicję:

Definicja

Asynchronous I/O is a form of input/output processing that permits other processing to continue before the transmission has finished.

Asynchroniczne I/O

Gdy zapytamy wikipedię, czym jest asynchroniczne I/O, to otrzymamy taką definicję:

Definicja

Asynchronous I/O is a form of input/output processing that permits other processing to continue before the transmission has finished.

Czy możemy zatem użyć asynchronicznego I/O w naszym programie?

Asynchroniczne I/O w pythonie

Spróbujmy dopisać słówko `async` do funkcji i ją uruchomić.

```
1  async def main():
2      return "hello, world!"
3
4  print(main())
```

Asynchroniczne I/O w pythonie

Spróbujmy dopisać słówko `async` do funkcji i ją uruchomić.

Tym razem dostaliśmy dwa ostrzeżenia oraz dziwny obiekt. Co starają się nam one przekazać i czym naprawdę jest nasza *„funkcja”*?

```
1  async def main():
2      return "hello, world!"
3
4  print(main())
```

```
async.py:4: RuntimeWarning: coroutine 'main'
was never awaited
  print(main())
RuntimeWarning: Enable tracemalloc to get
the object allocation traceback
<coroutine object main at 0x7f3c9d240200>
```

Asynchroniczne I/O w pythonie

Zobaczmy co robi przykład w dokumentacji modułu `asyncio`.

Asynchroniczne I/O w pythonie

Zobaczmy co robi przykład w dokumentacji modułu `asyncio`.

```python
import asyncio

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

asyncio.run(main())
```

Asynchroniczne I/O w pythonie

Zobaczmy co robi przykład w dokumentacji modułu `asyncio`.

Mamy tu:

- Jakiś specjalny `sleep()`
- Słowa kluczowe `async` oraz `await`
- Dziwne wywołanie funkcji za pomocą `asyncio.run()`

```
1  import asyncio
2
3  async def main():
4      print('Hello ...')
5      await asyncio.sleep(1)
6      print('... World!')
7
8  asyncio.run(main())
```

Asynchroniczne I/O w pythonie

Zobaczmy kolejny przykład.

Asynchroniczne I/O w pythonie

Zobaczmy kolejny przykład.

Mamy to dwa razy `asyncio.sleep`, więc program powinien się wykonywać 5 sekund.

```python
import asyncio


async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)


async def main():
    await asyncio.gather(
        say_after(2, "hello ..."),
        say_after(3, "... world"),
    )

asyncio.run(main())
```

Coś się wykonuje na raz...

Spróbujmy to zepsuć

Zmodyfikujmy sobie nasz przykład.

Coś się wykonuje na raz...

Spróbujmy to zepsuć

Zmodyfikujmy sobie nasz przykład.

```python
import asyncio
from time import sleep


async def say_after(delay, what):
    sleep(delay)
    print(what)


async def main():
    await asyncio.gather(
        say_after(2, "hello ..."),
        say_after(3, "... world"),
    )

asyncio.run(main())
```

Coś się wykonuje na raz...

Spróbujmy to zepsuć

Zmodyfikujmy sobie nasz przykład.

Słówko `await` robi jakieś rzeczy podobne do `yield`. Wykonywanie programu się zatrzymuje w środku *„funkcji”*, jednak tym razem nie widać co i gdzie kontroluje nasz program.

```
1  import asyncio
2  from time import sleep
3
4
5  async def say_after(delay, what):
6      sleep(delay)
7      print(what)
8
9
10 async def main():
11     await asyncio.gather(
12         say_after(2, "hello ..."),
13         say_after(3, "... world"),
14     )
15
16 asyncio.run(main())
```

Psucia ciąg dalszy

Czym jest ta linijka z `await asyncio.gather()`? Spróbujmy się jej pozbyć...

```
1  import asyncio
2
3
4  async def say_after(delay, what):
5      await asyncio.sleep(delay)
6      print(what)
7
8
9  async def main():
10     await say_after(2, "hello ..."),
11     await say_after(3, "... world"),
12
13 asyncio.run(main())
```

Psucia ciąg dalszy

Czym jest ta linijka z `await asyncio.gather()`? Spróbujmy się jej pozbyć...

Co się tu dzieje?

```
1  import asyncio
2
3
4  async def say_after(delay, what):
5      await asyncio.sleep(delay)
6      print(what)
7
8
9  async def main():
10     await say_after(2, "hello ..."),
11     await say_after(3, "... world"),
12
13 asyncio.run(main())
```

Czym są te funkcje, że musimy je tak dziwnie uruchamiać?

Tym razem do czynienia mamy z korutynami, czyli funkcjami których działanie może zostać przerwane.

```python
import asyncio


async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)


async def main():
    await asyncio.gather(
        say_after(2, "hello ..."),
        say_after(3, "... world"),
    )

asyncio.run(main())
```

Czym są te funkcje, że musimy je tak dziwnie uruchamiać?

Tym razem do czynienia mamy z korutynami, czyli funkcjami których działanie może zostać przerwane.

Wykonywanie jest przerywane kiedy używamy słówka `await`, lecz nie jest to taka pełna magia.

```
1  import asyncio
2
3
4  async def say_after(delay, what):
5      await asyncio.sleep(delay)
6      print(what)
7
8
9  async def main():
10     await asyncio.gather(
11         say_after(2, "hello ..."),
12         say_after(3, "... world"),
13     )
14
15 asyncio.run(main())
```

Czym są te funkcje, że musimy je tak dziwnie uruchamiać?

Tym razem do czynienia mamy z korutynami, czyli funkcjami których działanie może zostać przerwane.

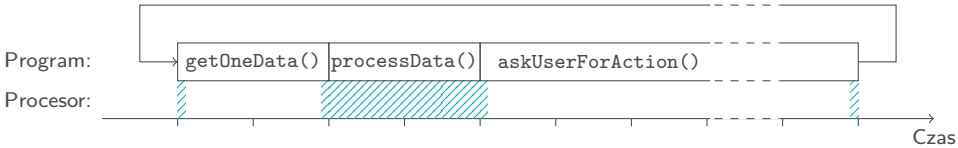
Wykonywanie jest przerywane kiedy używamy słówka `await`, lecz nie jest to taka pełna magia.

Nie możemy poprostu napisać wszędzie w naszym programie `async`/`await`, `asyncio` musi mieć na co czekać...

```
1  import asyncio
2
3
4  async def say_after(delay, what):
5      await asyncio.sleep(delay)
6      print(what)
7
8
9  async def main():
10     await asyncio.gather(
11         say_after(2, "hello ..."),
12         say_after(3, "... world"),
13     )
14
15 asyncio.run(main())
```

Jak możemy z asyncio skorzystać?

Spójrzmy ponownie na analizę wykonywania naszego programu.



Spróbujmy pobierać dane wtedy, kiedy czekamy na podjęcie decyzji przez użytkownika.

Pierwsze przepisanie

```python
import asyncio


async def main():
    data = await getOneDataAsync()
    while True:
        data = processData(data)
        # Get user input for previous data and fetch next data
        (new_data, user_decision,) = await asyncio.gather(
            getOneDataAsync(),
            askUserForActionAsync(data),
        )
        if user_decision:
            saveData(data)
        if new_data is None:
            # No data left
            break
        data = new_data

asyncio.run(main())
```

Pierwsze przepisanie

```python
import asyncio


async def main():
    data = await getOneDataAsync()
    while True:
        data = processData(data)
        # Get user input for previous data and fetch next data
        (new_data, user_decision,) = await asyncio.gather(
            getOneDataAsync(),
            askUserForActionAsync(data),
        )
        if user_decision:
            saveData(data)
        if new_data is None:
            # No data left
            break
        data = new_data

asyncio.run(main())
```

Czy możemy użyć tutaj paru generatorów?

Generatory oraz asyncio

`Asyncio` pozwala nam pisać asynchroniczne generatory, lecz nie możemy ich użyć bezpośrednio.

Możemy skorzystać z struktury kolejki zaimplementowanej przez `asyncio` by buforować dane między generatorem pobierającym dane oraz generatorem pytającym się użytkownika o decyzję.

```python
import asyncio

async def getData():
    while True:
        data = await getOneDataAsync()
        if data is None:
            # No data left to read
            break
        yield data

async def insertDataIntoQueue(queue):
    async for data in getData():
        processed_data = processData(data)
        await queue.put(processed_data)

async def getDataGenerator():
    queue = asyncio.Queue(maxsize=10)
    data_task = asyncio.create_task(insertDataIntoQueue(queue))
    while True:
        if queue.empty() and data_task.done():
            # No more data to process
            break
        yield await queue.get()

async def getUserActions():
    for data in getDataGenerator():
        if await askUserForActionAsync(data):
            yield data

async def main():
    async for data in getUserActions():
        saveData(data)

asyncio.run(main())
```

Podsumowanie

Asyncio pozwala nam na przerywanie wykonywania się programu podczas operacji wejścia/wyjścia. `Event loop` zarządza wykonującym się kodem asynchronicznym i wykonuje wtedy inne fragmenty programu.

Dziękuje za uwagę


STX NEXT