Programowanie
po 2 wykładzie

Andrzej Giniewicz

08.03.2023

Podczas ostatniego wykładu padły pytania o różnice pomiędzy iteratorami a generatorami, poniżej krótka notatka pomagająca rozjaśnić różnice pomiędzy tymi pojęciami.

1 Iteratory

Iterator to obiekt, który służy do przechodzenia pętlą po istniejącej strukturze. Najłatwiej zaobserwować ich działanie za pomocą funkcji `iter` i `next` oraz list.

```
lista = [3, 1, 2]
iterator = iter(lista)
print(next(iterator))
print(next(iterator))
print(next(iterator))
print(next(iterator))
```

W pierwszej linii kodu tworzymy listę. Alokuje to pamięć i tworzy listę na stercie. W kolejnej linii tworzymy iterator funkcją `iter`, czyli obiekt służący do przechodzenia listy element po elemencie. Funkcja `next` pobiera kolejny element. Możemy jej użyć trzy razy, przy czwartym użyciu zobaczymy wyjątek `StopIteration`. Po jednokrotnym odwiedzeniu każdego elementu iterator jest bezużyteczny.

Okazuje się, że gdy w Pythonie piszemy

```
for x in lista:
    print(x)
```

on w rzeczywistości wykonuje mniej więcej następujący kod (choć dodatkowo optymalizowany)

```
iterator = iter(lista)
try:
    while True:
        x = next(iterator)
        print(x)
```

```
except StopIteration:
    pass
```

Dzięki takiej implementacji pętla `for` może działać nie tylko z listami, ale też z innymi obiektami, dla których działają iteratory. Aby sam iterator mógł być przekazany do pętli, iterator iteratora powinien zawsze zwracać jego samego. Możemy sprawdzić, że tak się dzieje dla iteratorów list poniższym przykładem

```
lista = [1, 2, 3]
iterator = iter(lista)
iter(iterator) is iterator
```

co powierdza, że iterator iteratora jest nim samym (pamiętaj o różnicy pomiędzy tym samym a takim samym, tu sprawdzamy, że to ten sam obiekt operatorem `is`).

1.1 Generatory

Wiele iteratorów można zaimplementować podobnie jak ten omawiany w poprzednim podrozdziale, czyli za pomocą jednej metody `__next__`. Istnieje możliwość poproszenia Pythona, aby napisał iterator za nas. Ten sam efekt moglibyśmy uzyskać za pomocą specjalnej instrukcji Pythona `yield` oraz słowa kluczowego `def`. Jeśli we wnętrzu `def` zamiast `return` użyjemy `yield`, nie będzie to funkcja, tylko tak zwany generator.

```
def ones(n):
    while n > 0:
        n -= 1
        yield 1


for x in ones(10):
    print(x)
```

Pomiędzy klasą `Ones` i generatorem `ones` nie ma istotnej różnicy. Jak działają generatory? W momencie użycia funkcji `iter` Python uruchamia generator aż do momentu pierwszego wystąpienia `yield`. Wartość zwracaną przez `yield` odkłada na później i zwraca przy pierwszym wywołaniu funkcji `next`. Po wywołaniu funkcji `next` wznawia działanie generatora do kolejnego wystąpienia `yield`, który czeka w tle na następne wywołanie `next`. Python kontynuuje działanie w ten sposób, aż zakończy działanie `ones` bez natrafienia na `yield`. Wtedy zwraca wyjątek `StopIteration`.

`range` oraz `zip`, które omawialiśmy, działają w podobny sposób. Nie tworzą listy, tylko generator, który dopiero może zostać zamieniony na listę. Moglibyśmy stworzyć naszą własną implementację `range`.

```
def mój_range(a, b=None, c=1):
    if b is None:
        a, b = 0, a
    if c > 0:
```

```
        while a < b:
            yield a
            a += c
    elif c < 0:
        while a > b:
            yield a
            a += c
```

Kod ten odzwierciedla działanie funkcji range — zarazem jej wariant jedno, dwu oraz trzyargumentowy — zachowują się identycznie. Przykładowo poniższe wywołania są równoważne.

```
for x in range(3, 10, 2):
    print(x)

for x in mój_range(3, 10, 2):
    print(x)
```

1.2 Składanie generatorów

Podobnie jak listy, możemy również składać generatory. Wynikiem działania składania generatorów jest generator, czyli obiekt, który nie zajmuje tyle pamięci, co cała lista, a jedynie tyle, ile musi w danym momencie. Zwróćmy uwagę, że range pamięta tylko trzy liczby — a, b oraz c, a nie całą listę liczb.

Zapiszmy za pomocą składania list sumę liczb parzystych z listy.

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum([x for x in lista if x%2==0])
```

W rozwiązaniu tym jest pewien problem — Python tworzy nową listę 5 elementów (tyle w przykładzie mamy liczb parzystych), a następnie dodaje elementy na tej liście. Lista ta nie jest później potrzebna i zostanie skasowana, więc wykonujemy sporo zbędnych operacji. Przeciwnicy tego rozwiązania powiedzą wtedy, że choć zapis ten jest krótki i wygodny, to powolny, więc lepiej z niego nie korzystać. Okazuje się, że zamiast tworzyć listę liczb parzystych, możemy stworzyć generator, który wykona te same obliczenia, ale nie będzie zajmował dodatkowej pamięci na elementy.

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def parzyste(l):
    for x in l:
        if x%2==0:
            yield x
sum(parzyste(lista))
```

Niestety, straciliśmy czytelność. Okazuje się, że ten sam generator możemy zapisać za pomocą składania.

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum((x for x in lista if x%2==0))
```

Zwróćmy uwagę na różnicę pomiędzy rozwiązaniem ze składaniem list a generatorów — jedyna różnica to zmiana nawiasu kwadratowego na okrągły — ponieważ właśnie okrągłych nawiasów używamy do składania generatorów. Kod ten i wcześniejszy z `yield` są równoważne.

Ponieważ Python stara się zachować czytelność i nie korzystać ze zbędnych znaków, jeśli funkcja ma tylko jeden argument, przekazując jej generator składany jako argument, możemy pominąć dodatkowe nawiasy. W ten sposób właśnie działa `sum`, więc rozwiązanie da się zapisać jako

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum(x for x in lista if x%2==0)
```

Ostatecznie, co zrobić w rozwiązaniu

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum([x for x in lista if x%2==0])
```

aby zaczęło działać szybciej? Pragmatycznie możemy powiedzieć „skasować nawiasy kwadratowe". Niemniej jednak to, co wtedy robimy, ma głębszy sens — powoduje to zmianę kodu z iterowania list na iterowanie generatorów, zatem otrzymujemy wszystkie korzyści z tym związane — w szczególności brak konieczności alokowania pamięci na tymczasowe wyniki, czyli listę liczb parzystych w naszym przykładzie.

1.3 Generatory i iteratory nieskończone

Generatory i iteratory mogą być nieskończone. Ich definicja jest w pełni rozsądna i poprawna. Przykładowo generator poniżej zwraca naprzemiennie x oraz $-x$.

```
def naprzemian(x):
    while True:
        yield x
        yield -x
```

Generatora tego możemy używać, ale musimy uważać. Poniższy kod zwróci 42 oraz −42 kilka razy.

```
generator = naprzemian(42)
print(next(generator))
print(next(generator))
print(next(generator))
print(next(generator))
print(next(generator))
print(next(generator))
```

Pamiętajmy jednak, że generator ten nigdy nie zwraca `StopIteration`. Oznacza to, że gdy spróbujemy go użyć w pętli `for`, będzie ona pętlą nieskończoną, którą będziemy musieli przerwać instrukcją `break` w którymś momencie. Inaczej program nigdy nie przestanie działać. Nie możemy też zamienić generatora na listę komendą `list(naprzemian(42))`, ponieważ Python będzie tworzył coraz większą listę, aż skończy mu się pamięć przydzielona przez system operacyjny — nieskończone listy nie dadzą się zapisać w komputerze. Niemniej jednak takie nieskończone generatory mogą być użyteczne. Niekiedy jako nieskończony generator możemy zapisać na przykład interfejs do termometru za oknem. Za każdym razem, gdy użyjemy `next` zostanie pobrana temperatura. Oczywiście, pogoda nigdy się nie kończy, więc `next` możemy wywoływać w nieskończoność i nigdy nie zobaczymy wyjątku `StopIteration`.

1.4 Biblioteka itertools

W standardowej bibliotece języka Python znajduje się biblioteka `itertools`, która zawiera wiele przyjaznych i użytecznych iteratorów oraz funkcji do przetwarzania iteratorów[1].

[1] Dokumentacja biblioteki `itertools` znajduje się pod adresem `https://docs.python.org/3/library/itertools.html`.