**Listy składane. Generatory**

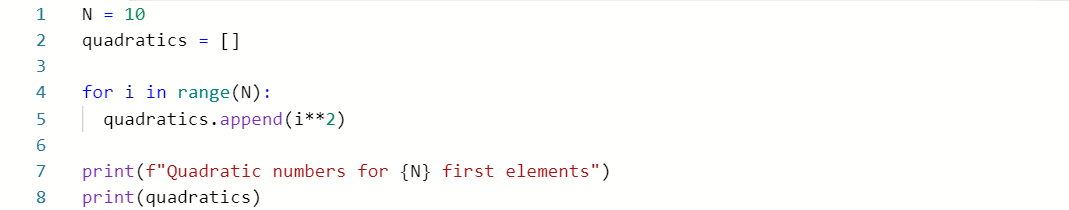
**Czym są listy składane?**

Python zapewnia zwięzłą składnię pozwalającą na utworzenie listy na podstawie innej. Te wyrażenia są nazywane **listami składanymi**.

Choć z doświadczenia wiem, że listy składane nie są raczej powszechnie wykorzystywanym tworem w komercyjnym kodzie produkcyjnym **(pewnie dlatego, że umożliwiają zmniejszenie ilości kodu, ale równocześnie ograniczają jego czytelność),** to warto mieć świadomość, czym one tak naprawdę są w Pythonie i jak z nich korzystać.

**Przykład 1:** *Obliczenie N-kolejnych kwadratów liczb naturalnych i umieszczenie ich w liście.*

Zadanie to możemy rozwiązać w standardowy sposób, wykorzystując pętlę for.

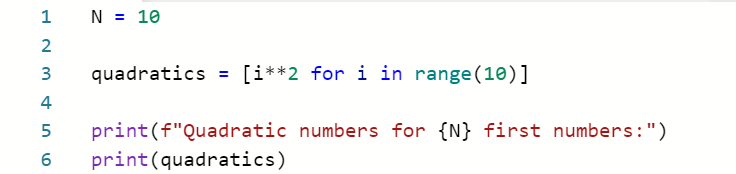


**Listing 1**



Aczkolwiek, jak już się pewnie domyślasz, Python dostarcza o wiele **krótszy sposób** na wykonanie powyższego zadania.

Spójrz, jak to może zostać rozwiązane z wykorzystaniem właśnie list składanych, (ang. list comprehension).



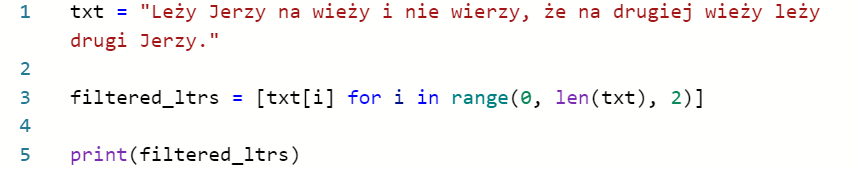
**Listing 2**

Zauważ, jak znacznie skrócił się Nasz kod (choć według mnie, jest też druga strona medalu - bo zmniejszyła się jego czytelność).

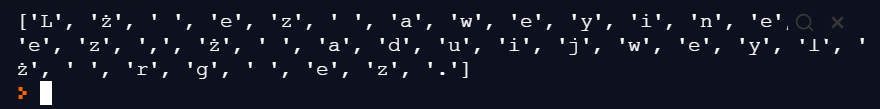
Analizując **linię 3.** listingu 2., można na pewno rzec, że dzięki niej upiekliśmy kilka pieczeni na jednym ogniu.

Zauważ, że wyrażenie listy składanej zarówno służy do **utworzenia nowej listy** oraz **inicjalizowania jej nowymi wartościami** – które są de facto obliczane wraz z kolejnymi iteracjami pętli for.

**Przykład 2:** *Umieść w liście co drugi znak z dowolnego zdania.*



**Listing 3**



**Syntax list składanych:**

Po przeanalizowaniu przykładów, możemy wysnuć ważny wniosek.

Zasada tworzenia list składanych ma się następująco:

**[wyrażenie for elem in elements],**

a po poznaniu dalszych zagadnień, nawet

**[wyrażenie for elem in elements if warunek]**

**Używaj list składanych zamiast funkcji map() i filter()**

Sekcję tę chcę poświęcić na główne zalety wynikające z korzystania list składanych. Jedną z nich już poznaliśmy – skrócenie kodu. A jak się to ma do programowania funkcyjnego (deklaratywnego)?

Jeżeli przerobiłeś już odpowiednie szkolenia, to na pewno kojarzysz takie funkcje jak **map()** oraz **filter().**

To, na co chciałbym zwrócić uwagę, to fakt, że gdy stosowana jest funkcja jednoargumentowa, w prostych przypadkach użycie listy składanej będzie czytelniejsze niż użycie wbudowanej funkcji!

* **map()**

Funkcja map() wymaga bowiem utworzenia funkcji lambda przeprowadzającej obliczenia, co niepotrzebnie komplikuje kod.

Wracając do **listingu 1.,** gdybyśmy chcieli funkcyjnie, otrzymać wartości kwadratowe danych liczb, moglibyśmy zapisać coś takiego:

**squares = list(map(lambda x: x \*\* 2, range(N)))**

Przyznaj, że już na tym etapie nie wygląda to dobrze.

A co w momencie, gdybyśmy wprowadzili dodatkowy warunek i chcieli obliczać **kwadraty tylko liczb parzystych**?

Wówczas musielibyśmy zapisać coś w podobie do:

**squares\_even = list(map(lambda x: x \*\* 2, filter(lambda x: not x % 2, range(N))))**

Tymczasem, z wykorzystaniem **comprehension lists**, ograniczymy się jedynie do:

**squares\_even2 = [i\*\*2 for i in range(10) if not i % 2]**

Chyba nie muszę udowadniać, co jest czytelniejsze i łatwiejsze do przeczytania? : )

**Instrukcje warunkowe w listach składanych**

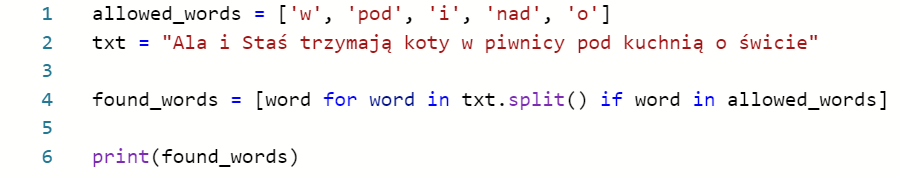
Z tego miejsca możemy sprawnie przemieścić się do sekcji opisującej łączenie **instrukcji warunkowych z listami składanymi** (de facto wykorzystaliśmy już tę własność, **obliczając squares\_even2 w przykładzie wyżej**).

Chcąc dodać instrukcje warunkowe w listach składanych, możemy zastosować zapis tak jak poniżej:

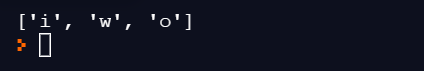
**[wyrażenie for elem in elements if warunek]**

W ten sposób będziemy budowali listę z elementów otrzymywanych przez **iterację** **i spełniających określony warunek**.

**Przykład 1:** Wyłuskaj z dowolnego zdania tylko dozwolone wyrazy.



**Listing 4**



To, co trzeba sobie przyswoić z przedstawionych przykładów to, że kod w listach złożonych **nie jest analizowany od początku lewej strony całego wyrażenia**.

Interpreter bowiem kolejno:

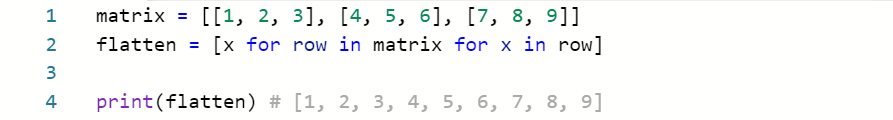
1. Pobiera do zmiennej word wyraz z **txt.split()** (for word in txt.split())
2. Sprawdza czy wyrażenie if jest prawdą (if word in allowed\_words])
3. Jeśli tak, zapamiętuje w liście rozpatrywany wyraz ([word).

**Unikaj więcej niż dwóch wyrażeń na liście składanej**

Aby nie było tak kolorowo, celem tej sekcji (jak zresztą możesz wywnioskować po tytule) jest zwrócenie uwagi na niepraktyczność list składanych przy **bardziej zaawansowanych przykładach**.

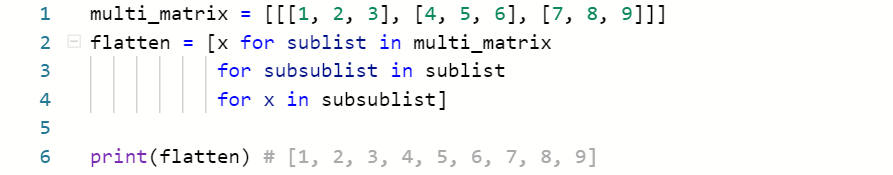
Poza prostymi przykładami użycia, listy składane obsługują również **wiele poziomów zapętlenia**. Przyjmujemy założenie, że…

**Przykład 1:** *Chcemy uprościć dwuwymiarową macierz do postaci pojedynczej listy zawierającej wszystkie elementy.*



Do tego momentu Nasz kod wygląda w porządku. Musisz przyznać, że zgrabnie rozwiązaliśmy problem. Jednak rozważmy przypadek, gdybyśmy musieli spłaścić macierz składającą się z **większej ilości zagnieżdżeń**…

**Przykład 2:** *Spłaszcz wielopoziomową listę*



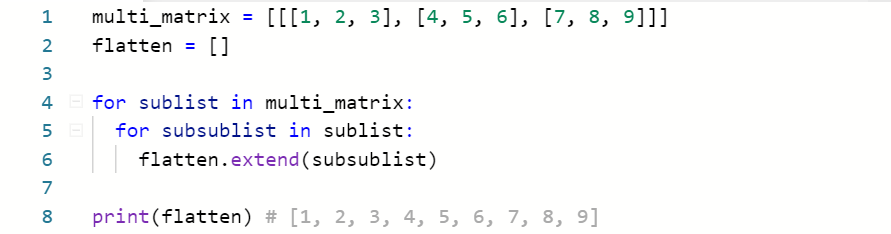
**Listing 5**

Jakość kodu nie jest już tak dobra jak we wcześniejszych przykładach.

Wadzi Nam bowiem **linia 2., 3., 4. i rozpostarta tak na nich długa lista składana.**

W momencie gdybyśmy chcieli dodatkowo wprowadzić sprawdzanie w niej dowolnych warunków, stworzylibyśmy jeszcze większego potwora!

A ostatecznie, żeby rozwiązać problem złożoności, wystarczyłoby zastosować standardowe podejście (z zagnieżdżonymi pętlami) i zapisać kod tak jak niżej:



**Listing 6**

**Wniosek?**

Listy składane mogą faktycznie upraszczać kod, ale do pewnego stopnia. Jak ze wszystkim - trzeba uważać, aby nie przesadzić.

Rozwijając jeszcze moją myśl z wstępu, powiem Ci, że zazwyczaj unika się używania list składanych w kodzie komercyjnym, niezależnie, czy przetwarzasz najprostsze przypadki, czy też wchodzisz w przypadku multi-zagnieżdżenia.

Wydaje mi się, że firmy w ten sposób zabezpieczają się zapewne przed dużym prawdopodobieństwem stworzenia zbyt złożonego kodu.

Na tak treściwym wniosku, chciałbym zakończyć temat związany z listami składanymi i omówić zupełnie nowy Pythonowy feature. Czas na generatory!

**Generatory**

Gdybym miał streścić w kilku słowach, czym są generatory, to powiedziałbym, że są to przede wszystkim specjalne funkcje, które pozwalają **optymalniej wykorzystywać pamięć operacyjną komputera**.

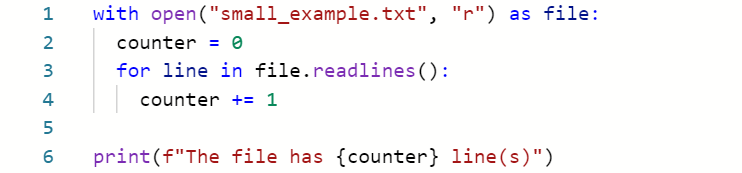
I jest to bardzo trafne spostrzeżenie, bo generatory są związane z tzw. **leniwymi iteratorami** (ang. lazy iterators).

W praktyce umożliwia to wyliczanie wyrażeń tylko wtedy, gdy następuje taka potrzeba i dzięki temu możemy efektywnie rozkładać w czasie kosztowne działania dla komputera.

Innymi słowy, kolejne rezultaty generatora (iteratora) będziemy pobierali tylko wtedy, gdy zajdzie taka potrzeba.

**Przykład 1:** *Odczyt dużych plików*

Załóżmy, że mamy następujące zadanie: **musimy zliczyć ilość linii w pliku**. W przypadku niezbyt obszernych plików, bez wahania moglibyśmy zrobić coś w podobie do:



**Listing 7**

Problem mógłby jednak pojawić się w momencie, gdyby plik był **ogromny i zawierał dużą ilość danych**.

Polecenie **file.readlines()** odczytuje bowiem całą zawartość w postaci listy do pamięci operacyjnej komputera! A pamięć RAM ma przecież ograniczenia co do pojemności.

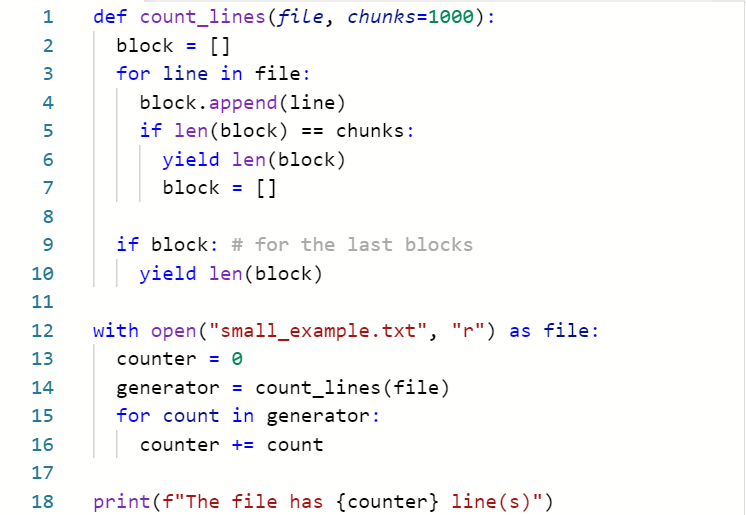
W momencie, gdybyśmy chcieli odczytać do pamięci duży plik (załóżmy że przekraczałby on rozmiar 4 GB) do pamięci o takiej pojemności, w trakcie działania programu natrafilibyśmy na wyjątek **MemoryError**.

Ponadto, przed rzuceniem wyjątku, program znacznie **spowolniłby działanie komputera** i mógłby nawet spowodować jego zawieszenie (sam tego doświadczyłem przy próbie opisania tego tematu).

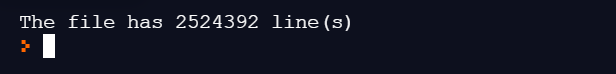
Sprytne rozwiązanie tego problemu mogłoby się ograniczyć do **wykorzystania generatorów.**

Generatory bowiem, tak jak wspomniałem wcześniej, są ściśle związane z leniwymi iteratorami i umożliwiają przechwycenie iteratora, spod którego pojedynczo wyłuskiwalibyśmy i naliczali kolejne linie pliku.

Na pewno lepiej wyobrazisz to sobie na przykładzie:

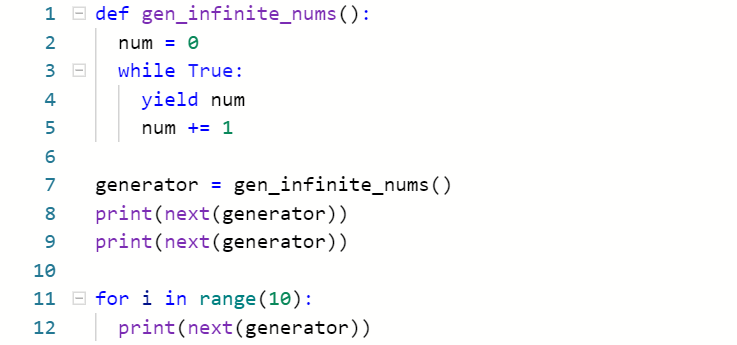


**Listing 8**



Tak, wiem, wiem. Mogłem zacząć od prostszego przykładu, ale wybacz mi to wyjście przed szereg. Chciałem zacząć od przytoczenia bardziej praktycznego przykładu. Niezależnie, jak bardzo klarowny jest dla Ciebie powyższy listing, przejdźmy do prostszego przykładu, który odzwierciedli praktyczne zastosowanie generatora.

**Przykład 2:** *Generowanie nieskończonego ciągu liczb.*



**Listing 9**



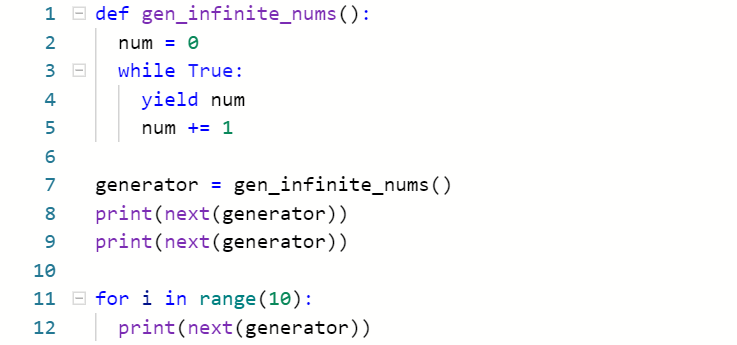
**O tajemniczym yield**

Na pewno zauważyłeś, że zarówno w listingu 8. i 9.,wykorzystaliśmy tajemnicze słowo **yield**.

To właśnie to słowo kluczowe rozróżnia **normalne funkcje od generatorów**.

Yield zachowuje się **podobnie** jak **return**, z tą różnicą, że **nie powoduje całkowitego wyjścia z generatora**, ale **wstrzymanie funkcji generatora i zwrócenie odpowiedniej wartości**.

Wracając jeszcze raz do listingu 9…



Linia 7. umożliwia stworzenie Nam generatora. To właśnie z niego będziemy korzystali, chcąc wyłuskiwać kolejne wartości wartości num zwracanego za pomocą słowa yield (linia 4).

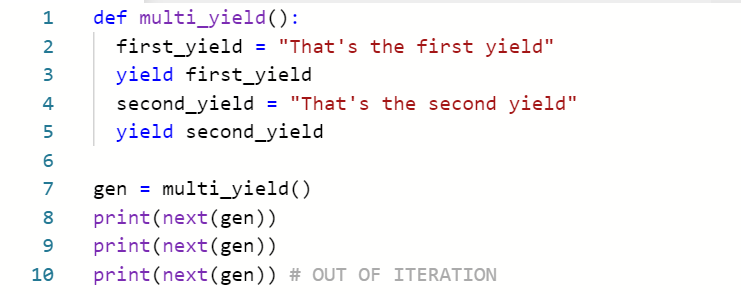
I tak, gdy wołamy next na stworzonym generatorze po raz pierwszy (linia 8), program wchodzi do jego funkcji (linia 1), wykonuje operacje, a gdy napotka yield (linia 4), wstrzymuje działanie i zwraca wartość w miejsce wywołania.

Przy kolejnych powtórzeniach next(), program **wraca dokładnie w miejsce**, **skąd wcześniej zwróciliśmy wartość** (linia 4) **i wznawia swoje działanie**.

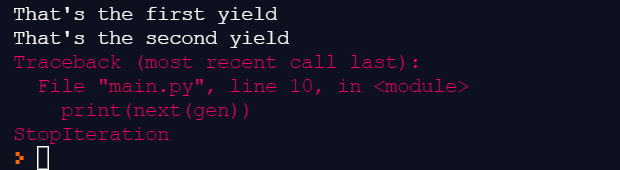
Tak więc wykonuje się linia 5 i z racji, że znajdujemy się w pętli, następuje kolejna iteracja while'a.

Zapamiętaj **- next** i **yield** to najczęściej pojawiające się słowa kluczowe w odniesieniu do generatorów. Next powoduje uruchomienie lub wznowienie funkcji generatora, yield natomiast ją wstrzymuje i zwraca odpowiednią wartość.

Przykład: Zobrazowanie procesu wstrzymywania - wznawiania programu.



**Listing 10**



**UWAGA:**

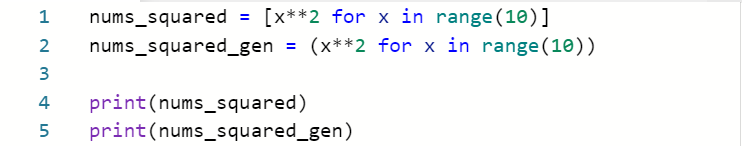
Zwróć uwagę na efekt (rzucony wyjątek **StopIteration**) w momencie, gdy próbujemy wywołać (wznowić) funkcję generatora po tym, jak zakończyła swoje działanie (linia 9. powoduje powrót do ostatniego yielda z linii 5. i w rezultacie opuszczenie funkcji generatora).

**Tworzenie generatorów przy użyciu wyrażeń**

Aby stworzyć generator, nie zawsze musimy **odwoływać się do funkcji generatora**. Czasami możliwe, a wręcz pożądane jest zapisanie wszystkiego w jednej linii.

Powiem Ci więcej, wyrażenia generatorów są **bardzo podobne do list składanych** i mogą wręcz przypominać - mylnie rozpoznawane - **tuple składane** (w Pythonie nie ma czegoś takiego jak tuples comprehension).

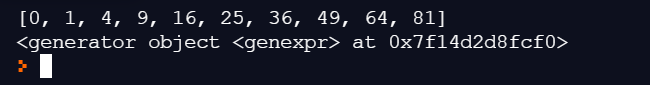
Przykład: Porównanie list comprehension (linia 1.) z generatorem (linia 2.)



**Listing 11**

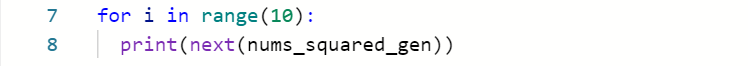
Różnica między linią 1. a 2. jest dość subtelna, ale bardzo znacząca na efekt działania.

Wynikiem poleceń zawartych w liniach 4 i 5 będzie bowiem:



Pierwsza linia outputu nie jest jakimś szczególnym zaskoczeniem, ale szczególną uwagę musimy zwrócić na to, że w linii 2. Utworzyliśmy generator!

Chcąc więc odwołać się do poszczególnych elementów, konieczny będzie poznany już zapis:



**Listing 12**

**Bardziej zaawansowane metody generatorów:**

W powyższych sekcjach pokryliśmy podstawy związane z generatorami. Chcąc jednak rozszerzyć Nasze horyzonty i wiedzę, musimy rozpatrzyć jeszcze kilka dodatkowych metod:

* **.send()**
* **.throw()**
* **.close()**

**Wysyłanie wartości do generatorów**

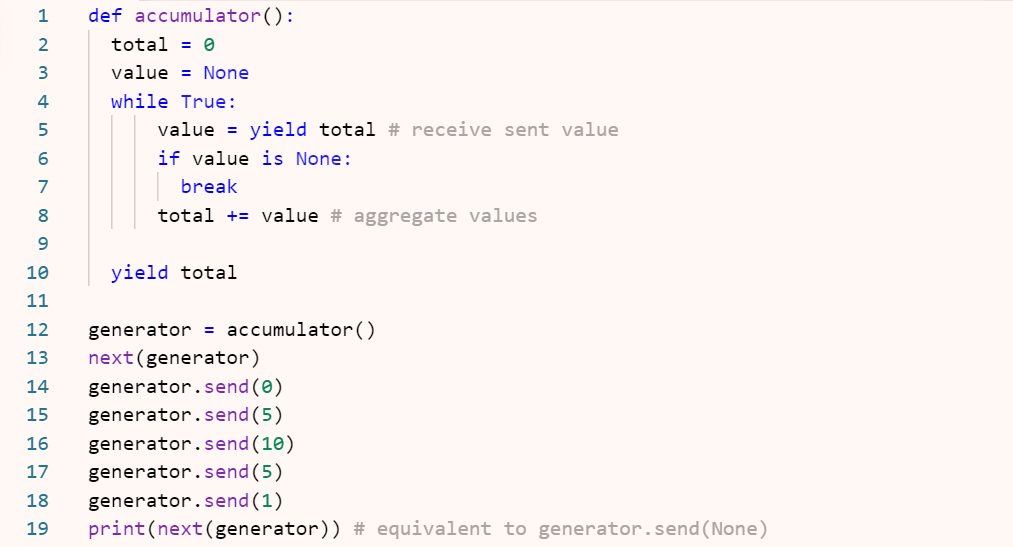
Metoda send wznawia generator i wysyła do niego wartość, która będzie użyta zaraz przy yield. Metoda ta bowiem powoduje zwrócenie pewnej wartości do yield’a i możliwość zapisania jej w zmiennej.

**Syntax:**

send() lub send(wartosc)

Z racji, że omawiana metoda umożliwia wysyłanie wartości do wznawianej funkcji generatora, możemy łatwo kontrolować przebieg programu!

**Przykład:** *Stwórz akumulator przesyłanych wartości:*



**Listing 13**



**UWAGA:**

Przed wysłaniem pierwszej wartości, konieczne jest wywołanie funkcji generującej tak jak w linii 13., aby mogła po raz pierwszy zatrzymać się na yield.

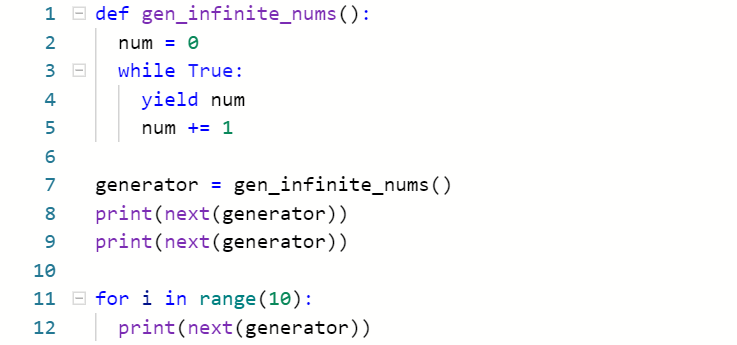
Polecenie next(generator) jest równoważne do generator.send(None)

**Throw() i close()**

Natomiast metody throw i close omówię równocześnie.

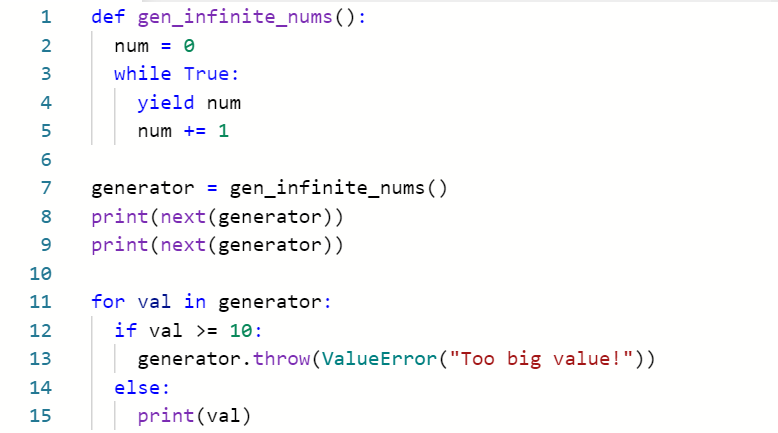
Throw() nakazuje generatorowi, aby jego funkcja, rzuciła wyjątek, a close(), aby przestała się wykonywać. Obie te funkcje ułatwiają sterowanie generatorem, ponieważ sami możemy określać moment zakończenia pracy funkcji.

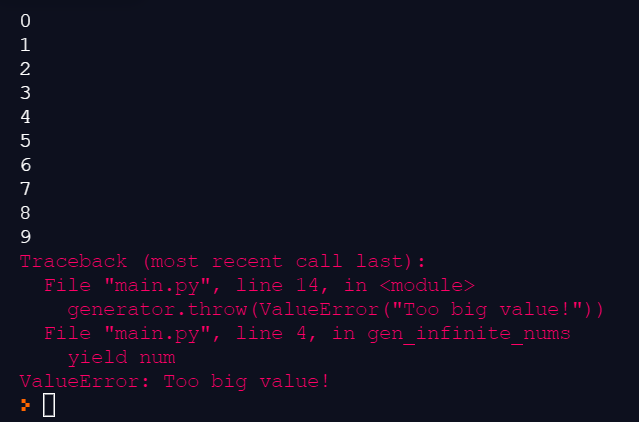
Wróćmy raz jeszcze do przykładu z generowaniem liczb z nieskończonego zakresu.



Możemy teraz zmienić zachowanie generatora w taki sposób, że po tym, jak num osiągnie pewną wartość, to rzucony zostanie wyjątek **ValueError** z stosownym komunikatem, np. **"Too big value".**

**Przykład:** *Wykorzystanie metody .throw()*

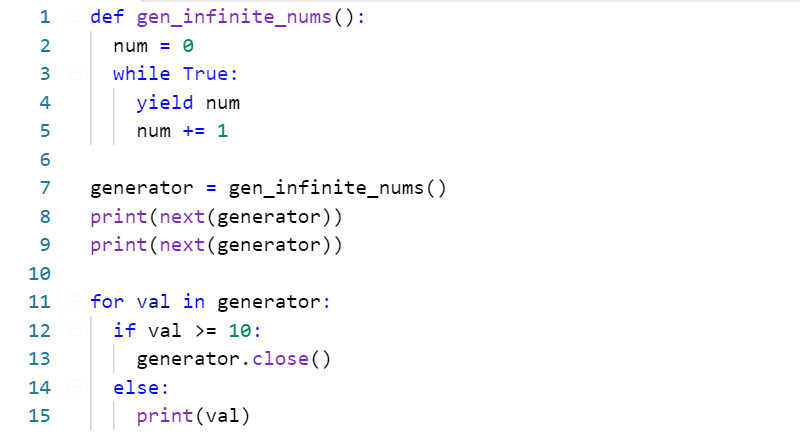


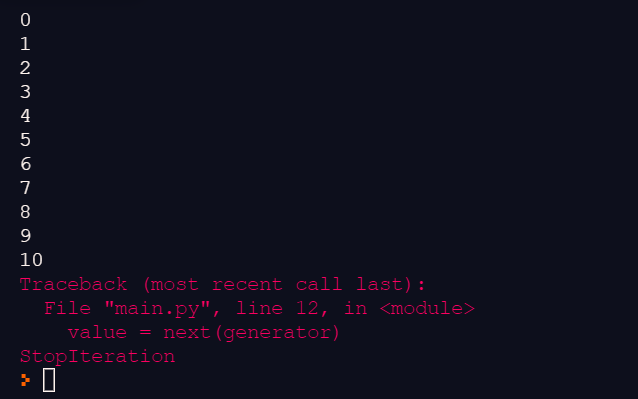


**UWAGA:**

Zauważ, że nie zawsze musimy wykorzystywać metodę next(), aby przechodzić po kolejnych wartościach zwracanych przez funkcję generatora. Czasami, chcąc pobierać wartości, dopóki funkcja generatora nie zakończy się (lub nie rzuci wyjątkiem), możemy zastosować zapis tak jak w linii 11 (przechodzenie pętlą po kolejnych elementach zwracanych przez funkcję.)

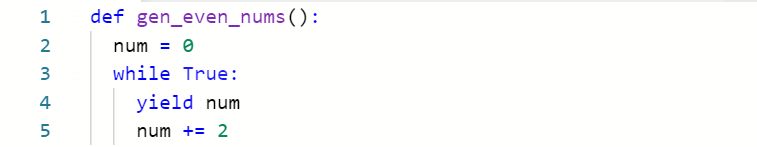
**Przykład:** *Zastąpienie .throw() metodą .close().*





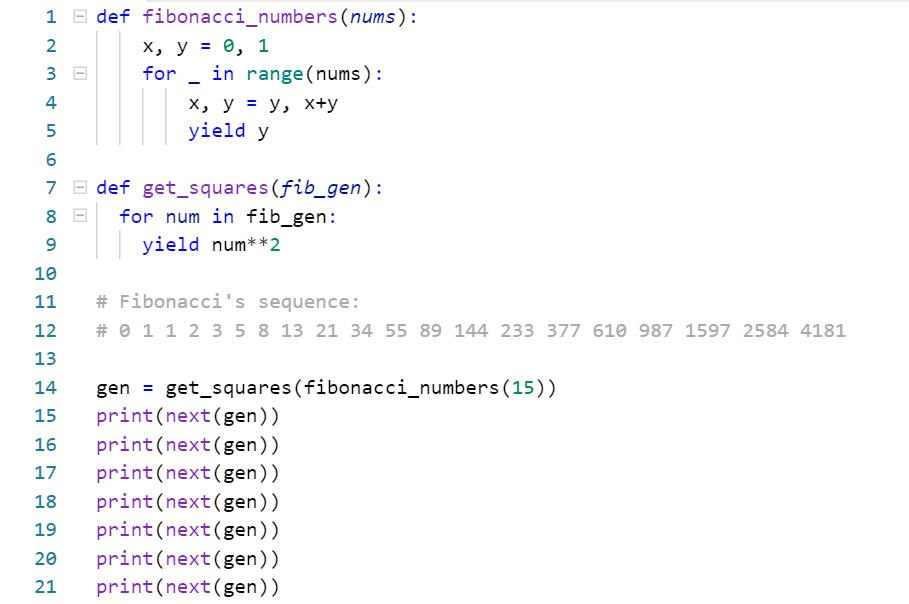
**Podsumowanie**

1. Generatory umożliwiają efektywniejsze zarządzanie pamięcią. Z racji wykorzystania przez nich zjawiska leniwej inicjalizacji, produkują tylko jeden konkretny obiekt i w przeciwieństwie do funkcji, w niektórych przypadkach, nie jest konieczne umieszczanie całej sekwencji obiektów w pamięci.
2. Generatory mogą służyć do tworzenia **nieskończonych strumieni danych** (infinite streams of data). Dobrym przykładem może być generowanie nieskończonej ilości liczb parzystych.



1. **Łączenie generatorów w pipeline’y**. Generatory mogą służyć również do łatwiejszego modyfikacji i manipulacji danymi.

Przykład: Załóżmy, że chcemy otrzymać ciąg składających się z **kwadratów kolejnych N wyrazów ciągu Fibonacciego**.



**UWAGA:**

Zwróć uwagę na to, jak funkcja generatora get\_squares zmienia elementy ciągu Fibonacciego. Po wywołaniu linii 14., program przesyła generator fibonacci\_numbers do funkcji get\_squares w postaci argumentu fib\_gen.

Pętla for (linia 8) przechodzi po kolejnych yieldowanch wartościach z poziomu fibonacci\_numbers, zwracając równocześnie kwadraty z poziomu linii 9. Wyniki te wrzucane są bezpośrednio do wywołań **next(gen)**.