Lista składana
List Comprehension - jest to konstrukcja, która umożliwia tworzenie listy przy użyciu zwięzłej składni. Jest to alternatywny sposób tworzenia listy w porównaniu do tradycyjnej pętli for.

Zapis wygląda następująco:

[wyrażenie for element in sekwencja warunek]

1. wyrażenie - zostanie obliczone dla każdego elementu w sekwencji.
2. element - przyjmuje wartości kolejnych elementów z sekwencji.
3. sekwencja - np. lista, napis, krotka przez którą iterujemy.
4. warunek (opcjonalny) t- warunek logiczny, który można zastosować do filtrowania elementów.

Przykład:

Stworzenie nowej listy – na podstawie znaków z ciągu znaków, tradycyjnie mogłoby wyglądać tak:

```
string = "Ala ma kota"
```

```
chars= []
for c in string:
    chars.append(c)
```

Za pomocą listy składanej:

```
chars = [ c for c in string]
```

Możemy też np. chcieć zliczyć litery w każdym słowie:

```
words = string.split()
word_counts = []
for w in words:
    word_counts.append( len(w))
```

I krótsza wersja:

```
word_counts = [ len(w) for w in words ]
```

W pierwszym przykładzie moglibyśmy zastosować filtr aby nie dodawać spacji do listy:

```
string = "Ala ma kota"
chars = [ c for c in string if c.isalpha()]
```

Inny przykład filtru, sprawdzenie, czy liczba podzielna przez 2:

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

Możemy wykorzystywać elementy wielu list:

```
list1 = [1, 2, 3, 4, 5]
list2 = [10, 20, 30, 40, 50]
result = [ e1 * e2 for e1 in list1 for e2 in list2 ]
print(result)
```

W tym wypadku przemnożyliśmy wszystkie elementy przez wszystkie. Jeżeli chcielibyśmy mnożyć elementy pod tym samym indeksem, można by użyć indeksu dla list:

```
result2 = [list1[i] * list2[i] for i in range(len(list1))]
print(result2)
```

*Co gdy będziemy mieli nierówną liczbę elementów?

Podobnie jak w wypadku list, możemy używać wyrażeń słownikowych:

{wyrażenie_klucz: wyrażenie_wartość for element in sekwencja}

```
keys = ['student1', 'student2', 'student3']
values = [5, 5, 4]
dictionary = {keys[i]: values[i] for i in range(len(keys))}
print(dictionary)
```

Iteratory i generatory

Iterator jest obiektem , umożliwiającym iterację przez elementy kolekcji, takiej jak lista, krotka, słownik, plik tekstowy a nawet obiekty naszej wlasnej klasy. Iterator przechowuje stan iteracji i udostępnia metody do pobierania kolejnych elementów z kolekcji, jeden po drugim.

Aby zdefiniować iterator, musimy zaimplementować dwie metody specjalne: **__iter__()** i **__next__()**. Metoda __iter__() zwraca samą siebie (iterator), a metoda __next__() zwraca kolejny element z kolekcji lub generuje wyjątek **StopIteration**, jeśli nie ma więcej elementów do zwrócenia.

Zwykła pętla for po liście może wyglądać tak:

```
lista = ["a", "b", "c"]
for e in lista:
    print(e)
```

Możemy wykorzystać wbudowany iterator dla listy:

```
iterator = iter(lista)
while True:
    try:
        e = next(iterator)
        print(e)
    except StopIteration:
        print("StopIteration")
        break
```

Albo napisać go samemu zgodnie z recepturą:

```
class ListIterator:
    def __init__(self, lista):
        self.lista = lista
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index >= len(self.lista):
            raise StopIteration
```

```
        else:
            e = self.lista[self.index]
            self.index += 1
            return e
```

Możemy stworzyć iterator dla własnej klasy (we własnej klasie) – przykład stosu z poprzednich zajęć:

```
class Stack:
    def __init__(self, max_size=5):
        self.items = []
        self.max_size = max_size
    def push(self, item):
        if len(self.items) < self.max_size:
            self.items.append(item)
        else:
            raise IndexError("Stack overflow")
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("Stack is empty")
    def is_empty(self):
        return len(self.items) == 0
    def size(self):
        return len(self.items)
    def __iter__(self):
        return self
    def __next__(self):
        if not self.is_empty():
            return self.pop()
        else:
            raise StopIteration
```

Generatory:

Generator w przeciwieństwie do list i ich iteratorów nie potrzebują przechowywać wszystkich danych w pamięci. Wykonują się w sposób leniwy.

```
generator = (x for x in range(10000))
size_generator = sys.getsizeof(generator)

list_comp = [x for x in range(10000)]
size_list_comp = sys.getsizeof(list_comp)
print("generator:", size_generator, "bajtów")
print("list comprehension:", size_list_comp, "bajtów")
```

W powyższym przykładzie użyliśmy wyrażenia generującego. Różni się od list składanych nawiasami.

Generator można utworzyć za pomocą funkcji generującej, automatycznie implementuje ona interfejs iteratora.

```
def even_numbers(max=):
    number = 0
    while True:
        yield number
        number += 2
```

wywołanie funkcji tak naprawdę zwróci generator:

```
gen = even_numbers()
print(type(gen))
```

Możemy na tym obiekcie użyć fuknkcji next().

Gdy używamy słowa kluczowego **yield** w funkcji generującej, automatycznie generowany jest kod, który obsługuje iterację i automatyczne wznawianie funkcji po każdym wywołaniu **next()**.

Kiedy w funkcji generującej zostaje wykonane polecenie **yield**, wartość jest zwracana jako rezultat, ale funkcja nie kończy swojego działania. Stan funkcji, włącznie z wartościami zmiennych lokalnych, jest przechowywany i może być wznowiony przy kolejnym wywołaniu funkcji next() na obiekcie generatora.

```
gen = even_numbers()
print(type(gen))
print(next(gen))
print("_____________")
gen = even_numbers()
for e in gen:
    print(e)
```

Jak widać możemy użyć generatora w pętli for.

Jeżeli np. chcielibyśmy mieć obiekt reprezentujący tytuły artykułów z wikipedii
https://www.kaggle.com/datasets/residentmario/wikipedia-article-titles
W tym wypadku lista 14846975 samych tytułów zajmuje 1145 MB.
Moglibyśmy wczytywać je wiersz po wierszu i utworzyć odpowiednie funkcjonalności – np. wczytanie artykułu o podanym temacie. Możemy też użyć generatora

```
def read_wiki_titles(filename):
    with open(filename, 'r') as file:
        for line in file:
            yield line.strip()
```

```
def read_titles(filename):
    with open(filename, 'r') as file:
        for line in file:
            yield line.strip()
title_generator = read_titles("small.txt")
for t in title_generator:
    print(t)
```

Artykuł z wikipedi możemy wczytać za pomocą Wikipedia-api:

```
import wikipediaapi
```

```
def get_article(title):
    w_api = wikipediaapi.Wikipedia('en')
    page = w_api.page(title)
    if page.exists():
        return page.text
    else:
        return ""
```

```
title_generator = read_titles("small.txt")
title = next(title_generator)
art = get_article(title)
print(art)
```

Zadanie:
Napisz generator, który użyje generatora read_titles oraz funkcji get_article do zwracania zawartości artykułów z wikipedii (plik small.txt – 1000 tematów).
Użyj tego generatora w funkcji, aby policzyć średnią liczbę występowania wszystkich odrębnych liter w artykułach.
Wywołaj funkcję i wyświetl średnią liczbę liter na artykuł.