Generatory

- Często jest tak, że potrzebujemy dużo wartości generowanych wg jakiegoś wzoru/przepisu, ale nie potrzebujemy ich wszystkich jednocześnie.
- W takich sytuacjach możemy skorzystać w Pythonie z iteratorów, bądź generatorów.
- Te pierwsze tworzone są jako klasy (i zajmiemy się nimi w dalszej części semestru), a drugie jako funkcje - i nimi się właśnie teraz zajmiemy.
- Jeżeli w ciele funkcji pojawi się słowo kluczowe **yield** to funkcja staje się generatorem.
- Po słowie **yield** może wystąpić obiekt, który jest z funkcji zwracany.
- Wystąpienie słowa **yield** oznacza zapamiętanie stanu funkcji, wyjście z niej i powrót do kodu, który ją wywołał.
- Kolejne odwołanie do funkcji powoduje powrót do instrukcji, która znajduje się bezpośrednio poniżej instrukcji **yield**, na której działanie funkcji zostało przerwane.
- Instrukcja **yield** może pojawić się w ciele funkcji więcej niż raz.
- Jeżeli nie ma już wartości, która z generatora miałaby być zwrócona zgłaszany jest wyjątek **StopIteration**, który jest automatycznie obsługiwany w pętli i skutkuje jej zakończeniem.

Przykład

```
def gen():
    x=1
    yield x
    x+=2
    yield x
    x+=2
    yield x



for i in gen():
    print(i, end=' ')     #1 3 5
```

Często instrukcja **yield** pojawiać się będzie wewnątrz pętli:

```
def gen(seq, f):
    for el in seq:
        yield f(el)
```

Często instrukcja **yield** pojawiać się będzie wewnątrz pętli:

```
def gen(seq,f):
    for el in seq:
        yield f(el)
```

```
w=[random.randrange(50) for _ in range(10)]
print(w)
#[9, 5, 3, 7, 28, 36, 8, 17, 38, 30]
```

```
for i in gen(w,lambda x: x**3):
    print(i, end=' ')
#729 125 27 343 21952 46656 512 4913 54872 27000
```

Często instrukcja **yield** pojawiać się będzie wewnątrz pętli:

```
def gen(seq,f):
    for el in seq:
        yield f(el)
```

```
w=[random.randrange(50) for _ in range(10)]
print(w)
#[9, 5, 3, 7, 28, 36, 8, 17, 38, 30]
```

```
for i in gen(w,lambda x: x**3):
    print(i, end=' ')
#729 125 27 343 21952 46656 512 4913 54872 27000
```

Powyższy generator możemy też zapisać korzystając z instrukcji **yield from**:

```
def gen(seq,f):
    yield from map(f,seq)
```

return w generatorze

- Instrukcja return chociaż może pojawić się w funkcji generatorowej, to jednak nie służy w niej do zwracania wartości a jedynie do ewentualnego przerwania jej działania, które skutkuje zgłoszeniem wyjątku StopIteration.

- Obecnie w funkcji generatorowej może pojawić się także instrukcja return z wartością, która stanowi komunikat powiązany z obiektem wyjątku.

return w generatorze

```
def gen(seq,f):
    for el in seq:
        if el>20:
            return 'blad'
        yield f(el)



w=[9, 5, 31, 7, 28, 36, 8, 17, 38, 30]

for i in gen(w,lambda x: x**3):
    print(i)                          #729,125
```

return w generatorze

```
def gen(seq,f):
    for el in seq:
        if el>20:
            return 'blad'
        yield f(el)
```

```
w=[9, 5, 31, 7, 28, 36, 8, 17, 38, 30]

for i in gen(w,lambda x: x**3):
    print(i)                          #729,125
```

```
g=gen(w,lambda x: x**3)
while 1:
    print(next(g))                    #729 125 StopIteration: blad
```

Zmiana stanu generatora

```
def gen(seq,f):
    while True:
        x = yield
        yield f(seq+x)


g=gen('Ala ma kota',lambda s: s.title())
next(g)
print(g.send('?'))      #Ala Ma Kota?
next(g)
print(g.send('!'))      #Ala Ma Kota!
```