Iteratory i generatory

- Niektóre obiekty są iterowalne (`list`, `set`, `str`, `range` etc.) – reprezentują ciągi obiektów.
- Po obiektach iterowalnych można iterować (`for`, listy/zbiory/słowniki składane, ...).
- Schemat iteracji: wzięcie elementu i jego *konsumpcja*.
- Termin „iteracja" odnosi się do całego procesu iterowania przez obiekt lub pojedynczego kroku tej iteracji – dla odróżnienia, pojedynczy krok będziemy dziś nazywać np. „obiegiem pętli" i podobnie.
- Niektóre obiekty iterowalne są konkretne (np. lista), niektóre wyliczane *leniwie* (np. `range`).
- Cel na pierwszą część wykładu: zrozumienie technicznej strony iteracji i pisanie własnych obiektów iterowalnych (np. iterowalny stos).

Iteratory i generatory

W iteracji uczestniczą w rzeczywistości **dwa** obiekty:

1. Obiekt iterowalny obj.
2. Jego *iterator*.

Przykład: analogia z Biblioteką i Bibliotekarzem [wykład/skrypt.
Uwagi do analogii:

- Nasza jedyna bezpośrednia interakcja z Biblioteką to wywołanie z niej Bibliotekarza (badanie książek z Biblioteki wykonuje się wyłącznie za jego pośrednictwem).
- Nie wiemy, w jaki sposób Bibliotekarz wynajduje odpowiedzi.
- Nie mamy gwarancji, że gdy rozpoczniemy nowy przegląd Biblioteki, to wyjdzie z niej ten sam Bibliotekarz. Biblioteka może być przeglądana przez wielu ludzi naraz, każdy obsługiwany przez innego Bibliotekarza.
- Wielu ludzi może pytać tego samego Bibliotekarza, a on nie odróżnia pytających – więc będzie zawsze podawał tytuł następnej książki.

Iteratory i generatory

W analogii, Biblioteka to obiekt iterowalny, a Bibliotekarz to jego iterator.
Schemat iteracji po obiekcie obj (nie tylko w Pythonie):

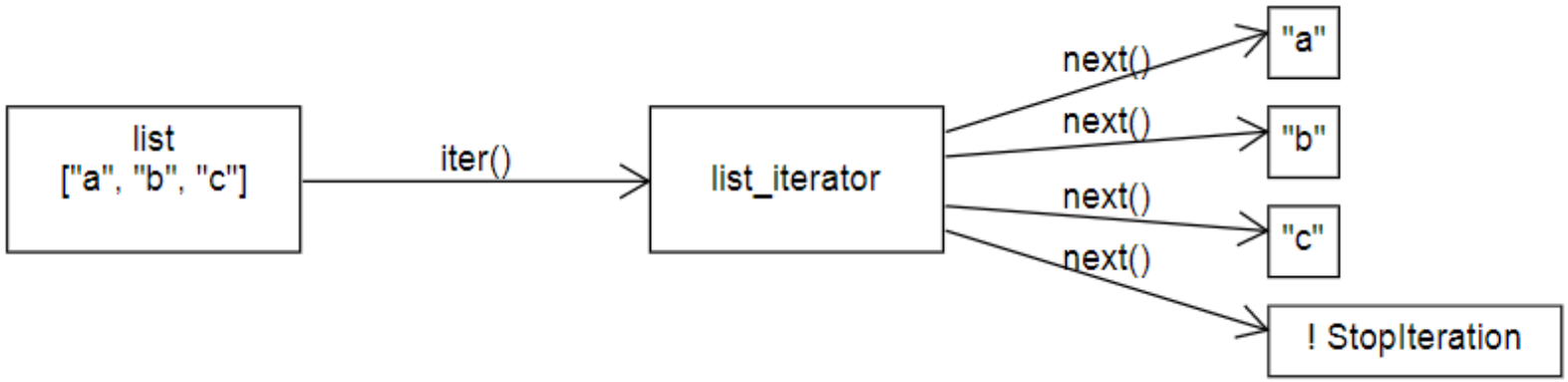
- *Konsument* (np. pętla) pobiera od obj jego iterator it. Za dostarczenie iteratora odpowiedzialny jest obj.
- Konsument pobiera z it „następny element". Jeśli takiego elementu nie ma, iterator to zgłasza i iteracja się kończy.
- Konsument konsumuje pobrany element i wraca do punktu (2).

W Pythonie mamy bezpośredni dostęp do operacji „pobranie iteratora" i „pobranie następnego elementu" przez wbudowane funkcje $\mathrm{iter}()$ i $\mathrm{next}()$.

Iteratory i generatory

„Ręczna” iteracja po liście:

```
>>> lst = ['a', 'b', 'c']
>>> it = iter(lst) # iterator listy
>>> it # list_iterator
>>> next(it) # 'a'
>>> next(it) # 'b'
>>> next(it) # 'c'
>>> next(it) # ! StopIteration
```



Pętle for, listy składane etc. realizują powyższe operacje (ukrywając przed nami szczegóły – pobieranie iteratora, elementów z iteratora, oraz obsługę wyjątku).

Iteratory i generatory

Operacje $\mathrm{iter(x)}$ i $\mathrm{next(x)}$ tłumaczą się na wywołanie „magicznych" metod $\mathrm{x.\_\_iter\_\_()}$ i $\mathrm{x.\_\_next\_\_()}$, odpowiednio.

(Ten sam mechanizm „magicznych metod" pojawił się już w obsłudze wielu wbudowanych operacji Pythona (np. $\mathrm{abs(x)}$ odpowiada $\mathrm{x.\_\_abs\_\_()}$, $\mathrm{x + y}$ odpowiada *zazwyczaj* $\mathrm{x.\_\_add\_\_(y)}$).

Aby obiekt obj był iterowalny, musi on realizować tzw. *protokół iteratorów*:

- Obiekt obj musi implementować metodę specjalną $\mathrm{\_\_iter\_\_()}$, zwracającą jego iterator.
- Zwrócony iterator musi implementować metodę $\mathrm{\_\_next\_\_()}$, która albo zwraca obiekt do skonsumowania, albo rzuca wyjątek $\mathrm{StopIteration}$, reprezentujący koniec iteracji; oraz metodę $\mathrm{\_\_iter\_\_()}$, zwracającą iterator (może nim być – i zazwyczaj jest – on sam).

Iteratory i generatory

Iterator posiada pewien stan wewnętrzny, który śledzi, na jakim kroku iteracji się znajdujemy.

Obiekt iterowalny może zwracać iteratory w dowolny sposób – najczęściej dla każdej iteracji tworzony jest osobny iterator, i stany iteratorów są niezależne:

```
>>> lst = ['a', 'b', 'c']
>>> it1 = iter(lst)
>>> it2 = iter(lst)
>>> it1 is it2 # False
>>> next(it1) # 'a'
>>> next(it1) # 'b'
>>> next(it2) # 'a'
>>> next(it2) # 'b'
>>> next(it2) # 'c'
>>> next(it2) # ! StopIteration
>>> next(it1) # 'c'
```

Iteratory i generatory

Iterator listy: wewnętrznie pamięta indeks listy, za każdym wywołaniem $\mathrm{next}()$ zwiększa go o 1 i zwraca obiekt spod tego indeksu. Stąd nieintuicyjne zachowanie kodu:

```
lst = list(range(6))
for x in lst:
    print(x)
    lst.remove(x)
print('lista po iteracji:', lst)
```

Którego efektem jest:

```
0
2
4
lista po iteracji: [1, 3, 5]
```

Iteratory i generatory

Iterator „bez kolekcji” (iteracja po nim samym):

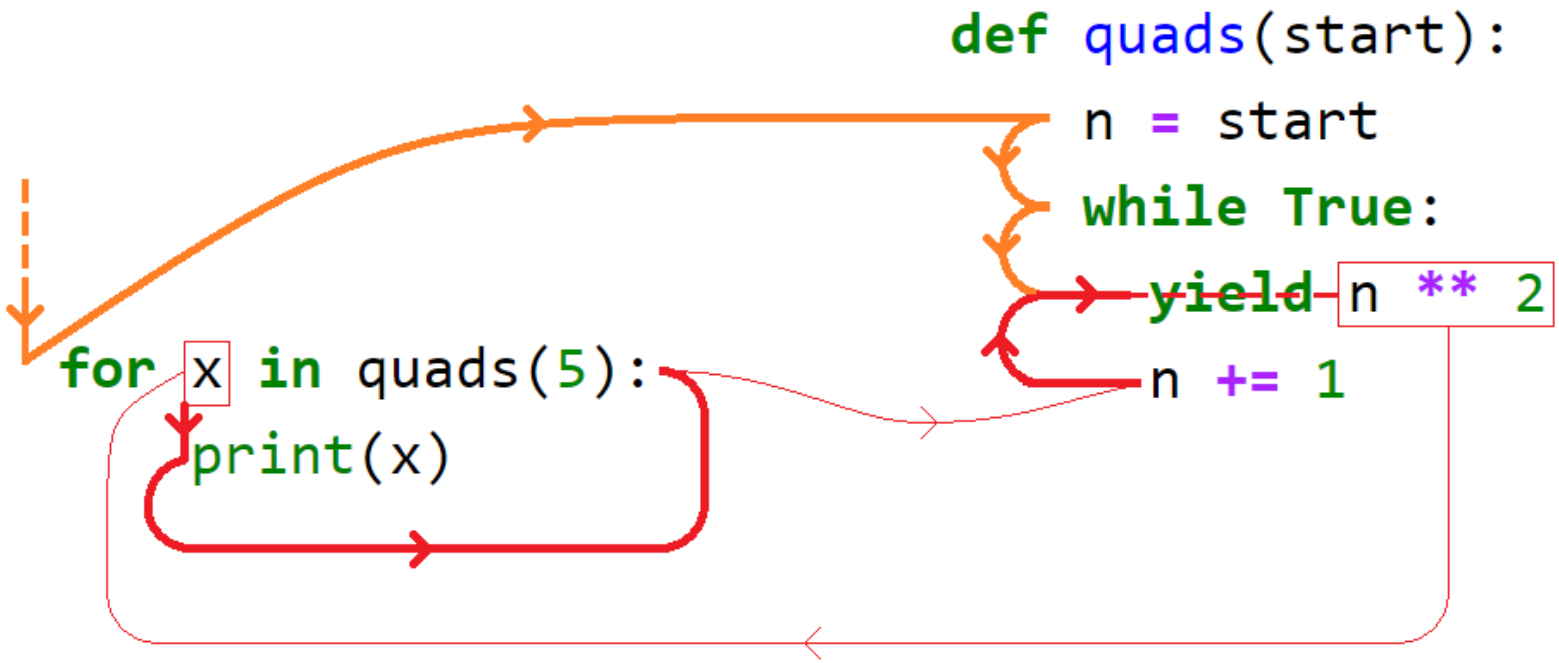
```
class Count:
    def __init__(self):
        self.n = -1

    def __next__(self):
        self.n += 1
        return self.n

    def __iter__(self):
        return self
```

Leniwa iteracja po (wszystkich) liczbach naturalnych:

```
for x in Count():
    print(x) # 0, 1, 2, 3, ...
```

Iteratory i generatory

Funkcje generujące [szerzej – wykład/skrypt]: słowo kluczowe yield sprawia, że wywołanie funkcji nie wykonuje jej, a tworzy *generator*, który zwraca obiekty zgodnie z treścią funkcji; yield zwraca obiekt i **zawiesza** działanie funkcji generujacej:



Wynik: 25, 36, 49, ...