przez referencję `ogon` do rekordu danych wskazywanego przez referencję `glowa` w zasadzie realizuje koncepcję zapętlonej listy jednokierunkowej!

- Jeśli chcesz, zachowaj też wskaźnik `poprzedni`, aby móc nawigować po takiej liście w obu kierunkach!

Listy z iteratorem

Nieco wcześniej pokazałem, jak bardzo możliwość bezpośredniego indeksowania upraszcza operacje na kolekcji danych, np. dzięki zapisaniu poszczególnych elementów w jakiejś tablicy. Odpowiednia interpretacja indeksów pozwala także na przechodzenie pomiędzy elementami kolekcji lub wyszukiwanie danych jest to możliwe dzięki zapisaniu danych nieuporządkowanych w „kontenerze" oferującym uporządkowany dostęp. Niestety, klasyczne kolekcje nieuporządkowane zapisane tak jak listy, jako zbiór obiektów w pamięci nie posiadają tablicowego odpowiednika indeksu i przechodzenie pomiędzy elementami kolekcji wymaga różnych karkołomnych realizacji i analizy elementów wskazywanych przez referencje dotyczące bieżącego elementu kolekcji (tutaj: listy). Można tu dostrzec zasadniczy problem, jakim jest próba użycia kolekcji danych służącej przede wszystkim do ich przechowywania do przeróżnych operacji o charakterze dostępowym, czasami sprzecznych ze sobą (np. wypisywania od początku, przeszukiwania według określonego kryterium, dynamicznego konstruowania podzbiorów listy spełniających pewien warunek).

Inaczej to formułując, powiedzmy, że potrzebny nam jest mechanizm oddzielający *użycie kolekcji* od jej *zawartości*. W tym celu nowoczesne języki programowania oferują koncept tzw. iteratorów, które są obiektami służącymi do swobodnego odnoszenia się do kolekcji danych. Iterator jest zatem pewnego rodzaju wskaźnikiem, który ułatwia pracę z kolekcjami obiektów. W typowej realizacji implementujemy go jako oddzielną klasę i używamy jako swego rodzaju nakładki na właściwy obiekt — kolekcję danych.

W rozdziale 8. omówię kilka realizacji klas-kolekcji dostępnych w Pythonie — typowo zawierają one wbudowane iteratory i ich użycie jest w zasadzie transparentne dla użytkownika. Nie dotyczy to jednak klas utworzonych przez programistę, który musi jawnie dołożyć obsługę iteratora! Celem tego punktu jest zapoznanie czytelnika z filozofią, jaka kryje się za tym pojęciem — iterator jest pewną *koncepcją*, która może mieć różnorodne, mniej lub (zazwyczaj) bardziej złożone — realizacje!

Iteratory są obecne w Pythonie w zasadzie... wszędzie, można ich użyć niejako **nie** np. w pętlach `for`, wyrażeniach inicjacji *list comprehension* lub **wprost** wywołać z klasy, która je wspiera, używając dedykowanych metod tego interfejsu.

Aby klasa została rozpoznana jako iterowalna, musimy w niej wbudować metodę o nazwie `__iter__()`. Celem tej metody jest zwrócenie obiektu klasy iteratora (napisanej osobno), która jest logicznie skojarzona z naszą klasą. Brzmi to

enigmatycznie, ale jak pokażę na przykładzie, nie jest zbyt skomplikowane w realizacji — szablon kodu jest dość prosty, oczywiście sama logika nawigowania po kolekcji może okazać się dość złożona, to już ściśle zależy od charakteru danej kolekcji!

Żeby nie dodawać nadmiernej ilości teorii do i tak rozległego tematu, jakim są iteratory, popatrzmy, jak się je realizuje w praktyce. Użyjemy do tego celu nieznacznie przerobionej listy jednokierunkowej opisanej na początku rozdziału w pliku *lista2.py* — skopiowałem jego fragment do skryptu *lista2iter.py*.

Jak zapewne pamiętasz, jej struktura miała postać (pomijam wcześniej rozwinięte fragmenty kodu opisane w punkcie „Dokładamy pierwsze metody użytkowe"):

```
class Element:
    def __init__(self, pDane, pNastepny=None):
        self.dane = pDane
        self.nastepny = pNastepny
# Dalej metody klasy 'Element':
    def wypiszElementy(self):
    ...
# ------------------------------------------------------------------------
class Lista:
    def __init__(self):
        self.glowa = None
        self.ogon = None
        self.dlugosc = 0
# Dalej metody klasy 'Element':
    def wstawNaKoniec(self, pDane):
    <...>
    def wypisz(self):
    <...>
    def szukaj(self, x):
    <...>
    def szukajRef(self, x):
    <...>
    def usunWybrany(self, x):
    <...>
```

Aby klasę `Lista` uczynić „iterowalną", należy na jej końcu dołożyć następującą metodę:

```
def __iter__(self):
    return MojIterator(self)
```

Zwraca ona obiekt klasy realizującej iterator — tutaj nazwałem ją `MojIterator` — i zostanie wywołana podczas próby iteracji przez obiekt klasy `Lista`.

Zerknijmy zatem na możliwą realizację klasy `MojIterator`. Ponieważ nasza kolekcja jest dość nieskomplikowaną listą jednokierunkową, którą możemy „przeiterować", począwszy od wskaźnika `glowa`, to kod inicjujący klasę może mieć taką postać:

```
class MojIterator:
    def __init__(self, pLista):
        self._kursor = pLista.glowa
```

Po prostu ładujemy do pola nazwanego kursor referencję do początku listy. Dalej, już podczas nawigowania po elementach kolekcji, kursor ten będzie się przemieszczał do kolejnych elementów listy. Czynność przemieszczania się jest realizowana za pomocą metody o specjalnej nazwie __next__(). Metoda ta będzie zwracała kolejny element kolekcji obsługiwanej przez iterator i wskazywany przez nasz umowny kursor:

```
def __next__(self):              # Zwraca kolejny element z kolekcji obsługiwanej przez iterator
    if self._kursor !=None:      # (*)
        res= "["+ str(self._kursor.dane)+"]"
        self._kursor=self._kursor.nastepny
        return res
    else:
        raise StopIteration      # Kończymy iterowanie
```

Metoda formatuje zwracaną na zewnątrz w zmiennej res wartość elementu kolekcji wskazywanego przez kursor i przemieszcza kursor do następnego elementu (*).

Uwaga: wymogiem składniowym w Pythonie jest wygenerowanie wyjątku StopIteration po osiągnięciu końca kolekcji!

Hm, jak na razie kod może nie jest jakiś spektakularnie atrakcyjny, ale możesz dzięki jego zaimplementowaniu użyć iteratora **w sposób niejawny**, np. w pętli for! Pokazuję to na przykładzie:

```
lista = Lista()  # Tworzymy pustą listę
for x in [1, 3, 5, 6, 12, 9]:
    lista.wstawNaKoniec(x)
print("Lista lista=", end=" ")
lista.wypisz()
print("Wywołujemy iterator poprzez użycie pętli 'for'")
for x in lista:
    print(x, end = " ")
```

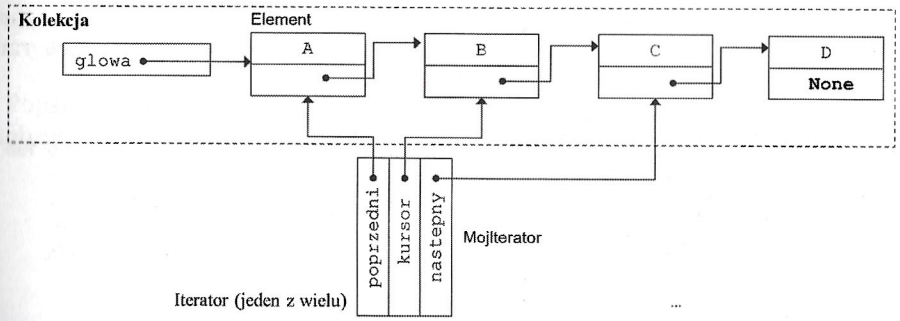
Wyniki:

```
Lista lista= 1 3 5 6 12 9
Wywołujemy iterator poprzez użycie pętli 'for'
[1] [3] [5] [6] [12] [9]
```

Gwoli formalności wspomnę, że w Pythonie możesz też ręcznie wywołać w sposób jawny metodę __next__(), ale jest to bardzo niewygodne i mało czytelne:

```
iterator = iter(lista) # 'lista' to zmienna zdefiniowana powyżej
while True:            # Pętla nieskończona
    try:               # Pobierz kolejny element:
        res=next(iterator)
        print (res, end= " ") # W tym miejscu przetwarzamy wartość wyłuskaną z kolekcji
    except StopIteration:
        break
```

Iterator może realizować dodatkowe funkcje, np. przechowywać dodatkowe wskaźniki niezdefiniowane w klasie bazowej (np. do elementu poprzedniego i następnego), co może nam pomóc w rozszerzaniu funkcjonalności klasy... poza definicją! Ten ostatni pomysł można graficznie przedstawić jak na rysunku 7.18.



RYSUNEK 7.18. Kolekcje z iteratorem — blok danych

Uff... trochę nam się tego kodu i pomysłów uzbierało, ale mam nadzieję, że czytelnik nie będzie miał kłopotu z ich analizą. Zastanówmy się teraz, jakie mogą być potencjalne korzyści wynikające z posiadania iteratora skojarzonego z kolekcją danych. Pierwszym nasuwającym się zastosowaniem może być swobodne przeglądanie kolekcji, połączone z pewnymi operacjami. Przykłady:

- Wyszukiwanie danych.
- Modyfikowanie kolekcji podczas wyszukiwania.
- Wstawianie połączone z sortowaniem (kursor zawsze będzie wskazywał na pewien rekord danych i np. korzystając z listy dwukierunkowej i w związku z tym dysponując kursorem mogącym się przesuwać w lewo i w prawo, znacznie szybciej znajdziemy prawidłowe miejsce na wstawienie!).
- Usuwanie według ustalonego kryterium — np. kasowanie co *k*-tego rekordu, tak aby pozostał na końcu określony element z kolekcji... — tak, mam na myśli sposób rozwiązania problemu Józefa Flawiusza przy użyciu listy cyklicznej i algorytmu wyszukująco-kasującego!

Każda z przedstawionych w tym rozdziale list ma swoje wady i zalety oraz ułatwia sprostaniu określonym wyzwaniom programistycznym. Niektóre listy określone jako nietypowe wcale nie są aż tak nietypowe, gdyż umożliwiają łatwiejsze rozwiązanie pewnych zagadnień algorytmicznych niż za pomocą innych struktur danych — mam nadzieję, że na podstawie pokazanych przykładów dotyczących list zorientujesz się, jakie techniki programowania warto stosować podczas modelowania własnych klas realizujących złożone funkcje biznesowe!

Zbiory

Odsapnijmy teraz nieco i zakończmy rozdział omówieniem bardzo prostego typu danych — zbioru. Dodajmy w tym miejscu, że Python jest tu w uprzywilejowanej pozycji, gdyż wspiera taki typ danych, ale spróbujmy go zrealizować niejako od zera i w wersji dopasowanej do naszych potrzeb!