- Realizacja struktury danych jako kontenera zwalnia programistę z troski o bezpieczeństwo zapisywania danych i zarządzania pamięcią.
- Do „przechadzania się" po kolekcjach służą wygodne iteratory, czyli znane nam już swego rodzaju obiektowe wskaźniki do elementów zapisanych w środku.
- Klasy często oferują wbudowane gotowe algorytmy, np. sortowania, wyszukiwania, zamiany, złączania.

Z punktu widzenia programisty użycie wybranej klasy jest proste, wystarczy tylko wbudować w kod odpowiednie instrukcje i użyć dyrektywy `import`.

Jaka jest cena używania gotowych klas? Zazwyczaj stosujemy je w bardzo prostym zakresie, wykorzystując mały fragment ich możliwości. Podobnie z kwestią zarządzania pamięcią — mamy tu nikłą kontrolę nad faktyczną realizacją alokacji obiektów. Ten koszt w przypadku struktur listowych jest stały z uwagi na samą naturę tego typu danych, ale już w tablicach dynamicznych, które rozszerzają się samoczynnie w miarę potrzeb, złożoność rośnie liniowo w przypadku próby włożenia czegoś poza bieżący zakres lub w środku — cudów nie ma, po prostu jakoś trzeba te klocki poprzesuwać, aby zrobić miejsce dla nowego. W związku z tym np. biblioteka NumPy napisana w C++ oferuje własne tablice, zoptymalizowane pod kątem szybkości dostępu i zarządzania pamięcią!

Patrząc na moje przykłady, na pewno nauczysz się korzystać z ważniejszych klas Pythona bez wnikania w różne niuanse związane z ich realizacją lub cechami. Pamiętaj, że to tylko wstęp — aby nauczyć się programowania w tym języku, warto sięgnąć do przeznaczonych do tego podręczników lub wymienionych dalej stron internetowych.

Co zatem zostanie opisane tym rozdziale i w jakim celu?

W dalszej części rozdziału omówię wybrane złożone typy danych, które są wbudowane w Pythona i pozwalają na przechowywanie zbiorów lub danych, realizujące mniej lub bardziej uporządkowane kolekcje danych (np. listy, słowniki...).

Takie typy są niezbędnym budulcem służącym do realizacji tzw. abstrakcyjnych typów danych (przypomnę: takich, które nie są wbudowane w język i pozwalają modelować skomplikowane systemy informatyczne) i ułatwiają realizację bardziej złożonych projektów informatycznych. Podsumowując, mój opis będzie dotyczył tylko tych klas i modułów, które będą nam przydatne w trakcie dalszej lektury.

Listy, czyli tablice dynamiczne

Wiesz już, że w Pythonie tablice nie występują jako odrębny byt, gdyż ich rolę przejęły *listy*, które oferują takie same możliwości (zapis sekwencji danych) a...

dostępowe. Oznacza to, że wszelkie tablice można zrealizować za pomocą tzw. list, które są uogólnioną strukturą danych, umożliwiającą również modelowanie zwykłych, klasycznych tablic.

Wcześniej pokazywałem wiele przykładów kodu Pythona, ilustrując jego wszechstronne możliwości. W jednym z plików została użyta następująca konstrukcja:

```
jezyki    = ["C++", "Python", "Java", "Lisp"]
```

Jej cechą charakterystyczną jest użycie nawiasów klamrowych (tzn. [oraz]), które wskazują na utworzenie tzw. **listy**.

Możliwe jest też utworzenie listy pustej i konstruowanie jej zawartości na bieżąco, np. na podstawie danych zawartych w innych zmiennych, po wczytaniu informacji z pliku dyskowego lub po wpisaniu przez użytkownika danych w komendzie `input()`. Oto przykładowe deklaracje tworzące listy *puste*:

```
l1=[]
l2=list() # Konstruktor klasy 'list'
```

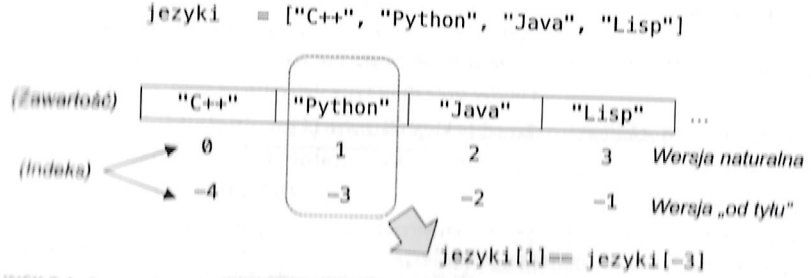
Jeśli chcesz zasymulować utworzenie tablicy o rozmiarze *N*, wypełnionej np. samymi zerami, to użyj konstrukcji podobnych do:

```
N=20
tab1=[0]*20
tab12= [x for x in range(N)]
print(tab1)    # Tworzy: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(tab12)   # Tworzy: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Oczywiście wartość i typ danych możesz dobierać według konkretnych potrzeb i mogą to też być wartości `None` albo obiekty klas!

Możesz także utworzyć listę o pewnej długości i powielając ją, zbudować tablice wielowymiarowe (opisałem to w rozdziale 5., w punkcie „Inicjalizacja tablic o stałym wymiarze").

Przyjrzyjmy się na przykładzie, w jaki sposób deklarujemy i jak następnie używamy list w Pythonie. Na rysunku 8.1 pokazano deklarację wzmiankowanej wcześniej listy `jezyki` i zasady dostępu do jej poszczególnych elementów.



RYSUNEK 8.1. Czteroelementowa (na razie!) lista w Pythonie

Lista została zainicjowana wstępnie czterema elementami i dostęp do [illegible] z nich jest możliwy w sposób, nazwijmy to, naturalny (indeks w nawiasach kwadratowych startujący od zera) lub „od tyłu” (indeks pozycji ostatniej to -1). W [illegible] tym można stwierdzić, że zmienne jezyki[1] i jezyki[-3] wskazują na ten [illegible] obiekt w pamięci!

Indeksowanie „ujemne” stanowi jeden z elementów „udziwniających” Pythona [illegible] szczycącego się swoją prostotą. Nie musisz intensywnie stosować tej notacji [illegible] dla Ciebie mało czytelna, ale na pewno przydaje się ona podczas konstruowania pętli [illegible] zliczających wstecz.

Tak gwoli ścisłości dodam, że rysunek upraszcza rzeczywistość, gdyż [illegible] sugeruje, że lista zawiera w sobie elementy, tak jakby to były pudełk[illegible] (tutaj: napisy). Tymczasem fizyczna realizacja listy w Pythonie poleg[illegible] dowaniu listy obiektów, ale *referencji do obiektów*. Jako użytkownik [illegible] jednak aż tak głęboko wnikać w szczegóły implementacyjne!

Popatrz na tabelę 8.1 podsumowującą cechy charakterystyczne list w Pythonie.

TABELA 8.1. Listy w Pythonie — cechy charakterystyczne

CECHA	KOMENTARZ
Uporządkowanie	Zapis sekwencyjny oznacza, że elementy są na liście [illegible] w sposób uporządkowany i można do nich sięgnąć [illegible] indeks, czyli numer porządkowy (pamiętaj, że numeracja [illegible] się od 0, a nie od 1).
Duplikaty	Dozwolone jest tworzenie listy zawierającej te same [illegible]
Modyfikowalność	Do listy można swobodnie dokładać elementy oraz [illegible] bez kłopotu można też podmieniać elementy w miejscu [illegible] jak to było z napisami w Pythonie? W nich nie można [illegible] żadnych elementów, ale... napisy w Pythonie nie są *listami* [illegible] tylko odrębnym typem danych!).
Dowolna długość (w praktyce)	Po wstępnej inicjacji możesz dokładać nowe elementy [illegible] ograniczonym w zasadzie wyłącznie dostępną pamięcią [illegible][1]
Dowolna zawartość	Listy mogą zawierać w zasadzie wszystko, co jest [illegible] Pythonie — znaki, liczby, złożone obiekty, a nawet [illegible] Dane w liście mogą być różnego typu (można mieszać [illegible] i liczby oraz inne typy danych).
Symbol rozpoznawczy	Nawiasy kwadratowe: [ ].

[1] W zasadzie minąłem się z prawdą, pisząc „dowolna długość”. W Pythonie [illegible] realizacji ogranicznikiem możliwości adresowania danych jest architektura [illegible] moim komputerze ta maksymalna długość wynosi 9 223 372 036 854 77[illegible] wywołanie sys.maxsize).

Metody dostępne dla list w Pythonie

Jakie operacje są dostępne dla zmiennych wskazujących na listy? W tabeli 8.2 podsumowałem podstawowe operacje wykonywane na listach Pythona.

TABELA 8.2. Podstawowe operacje wykonywane na listach w Pythonie

CECHA	KOMENTARZ

[] Operator dostępu	Operator dostępu [n] pozwala uzyskać dostęp do n-1-tego elementu (numeracja zaczyna się od zera!). Element musi jednak istnieć, w przeciwnym razie Python zgłosi wyjątek IndexError. Przykład:
	`jezyki = ["C++", "Python", "Java", "Lisp"]`
	`print (jezyki[1])`
	Wypisze *Python*, ale już `print (jezyki[7])` przerwie działanie skryptu wyjątkiem `IndexError: list index out of range`.

Operator + (tzw. konkatenacja)	Operator + pozwala skleić dwie lub więcej list i zwraca nową, np.: dla a=[2, 3] oraz b=['a',1] komenda print(a+b) wypisze [2, 3, 'a', 1].
Operator * (powielanie)	Operator * pozwala *powielić* listę; np. kontynuując powyższy przykład: `print(a*3)` wypisze [2, 3, 2, 3, 2, 3]. Jego użycie jest jednak ryzykowne, gdyż powielanie list wcale nie oznacza tworzenia nowych elementów w pamięci — Python zduplikuje tylko referencje do nich (ten fenomen opisałem w rozdziale 5.).
len (x)	Bardzo przydatna funkcja zwracająca długość listy x. Pomaga unikać błędów dostępu poza zakresem i kłopotliwego obsługiwania wyjątków.
in	Słowo kluczowe in pozwala na sprawdzenie istnienia elementu w liście. Przykład operujący znaną nam już listą jezyki: `print("Visual Basic" in jezyki)` wypisze False.

Ponieważ operator dostępu [] wymaga podania poprawnego numeru indeksu, to jeśli chcesz uniknąć obsługi wyjątku IndexError, musisz po prostu używać funkcji len() i adresować wyłącznie dostępne pozycje.

Python oferuje wiele użytecznych metod pozwalających „wycisnąć” z list niemal wszystko: dodać lub usunąć elementy, odszukać dane na liście, przesortować zawartość (tabela 8.3).

Używając list, warto sobie uświadomić pewne oczywiste *ograniczenia wydajnościowe*:

- O ile samo dokładanie na koniec listy jest bardzo szybkie, to już operacja index() może działać wolniej, gdyż w najgorszym przypadku wymagane jest przejrzenie całej listy!
- Podobnie usunięcie wybranego elementu przez remove() może być wyraźnie wolne w przypadku dłuższych list, gdyż konieczne jest przebudowanie całej listy.
- Czas sortowania jest liniowo proporcjonalny do długości listy.

TABELA 8.3. Metody dostępne dla list w Pythonie

CECHA	KOMENTARZ

`append(x)`	Dodaje element *x* na koniec listy. Przykład:
	`liczby=[3,6,8]`
	`liczby.append(10)`
	`print("Tablica 'liczby:'", liczby)`
	wypisze `Tablica 'liczby': [3, 6, 8, 10]`.

`clear()`	Kasuje (czyści) zawartość listy.
`count(x)`	Zwraca liczbę wystąpień elementu *x* w liście. Przykład: `a=[2, 3, 4, 2, 2, 5, 8, 9]` `print( a.count(2) )` wypisze *3*.
`extend(d)`	Rozszerza listę o elementy zawarte w *d*. Przykładowo, dla `a=[2, 3, 4]` i `b=[80, 90]` operacja `a.extend(b)` utworzy listę `[2, 3, 4, 80, 90]`.
`index(x)`	Zwraca indeks poszukiwanej wartości *x* lub wyjątek `ValueError`, gdy nie odnajdzie elementu *x*. Przykładowo, dla `a=[2, 3, 4, 2, 2, 5, 8, 9]` instrukcja `a.index(8)` zwróci *6*.
`insert(n, x)`	Wstawia element *x* przed indeksem *n*. Przykładowo, dla tej samej listy co wyżej operacja `a.insert(2, 200)` utworzy listę `[2, 3, 200, 4, 2, 2, 5, 8, 9]`.
`pop(n)`	Usuwa i *zwraca element* obecny na zadanej pozycji *n* (jeśli nie podasz parametru, to usuwany jest *ostatni* element, tj. z pozycji *–1*). Python zgłosi wyjątek `IndexError`, jeśli lista okaże się pusta lub indeks będzie spoza dozwolonego zakresu wartości danej listy. Przykład: dla `a=[10, 20, 30, 40, 8, 60]` operacja `a.pop(4)` zwróci *8*.
`remove(x)`	Usuwa pierwsze wystąpienie wartości *x* lub zgłasza wyjątek `ValueError`, gdy nie odnajdzie tego elementu.
`reverse()`	Odwraca listę *w miejscu*. Dla `a=[2, 3, 4, 2, 2, 5, 8, 9]` operacja `a.reverse()` *zmieni* pierwotną listę *a* na `[9, 8, 5, 2, 2, 4, 3, 2]`.
`sort()`	Sortuje listę w kierunku wartości rosnących (wersja domyślna, bez parametrów) lub w kierunku wartości malejących (wówczas wywołanie musi przybrać postać: `sort(reverse=True)`). Do metody `sort` można też przekazać drugi, opcjonalny argument, `key`, wskazujący na *funkcję* realizującą kryterium sortowania.

W kolejnym rozdziale opiszę realizację struktury zwanej stosem, nieco podobnej do listy, ale narzucającej ograniczenie dostępu polegające na nakładaniu elementów na siebie (nie można sięgnąć do elementu bez zdjęcia tych pozostałych, które były po nim dołożone).

Listy tworzone na podstawie wyrażeń

Python oferuje wiele intrygujących składniowo wyrażeń, które zwięźle zapisane (1 – 2 linijki kodu) często czynią cuda. Jednym z nich jest możliwość inicjalizacji listy na podstawie wyrażenia zawartego w nawiasie, mieliśmy już z tym mechanizmem wiele razy do czynienia.

Taki sposób inicjalizacji list w literaturze anglojęzycznej określany jest terminem *list comprehension*. Popatrz na prosty przykład kodu, który wytłumaczy ten mechanizm lepiej niż formalne definicje:

```
x1=[128, 64, 32, 16, 8, 4, 2, 1, 0]
x2 = [ n*n for n in x1 if n<=16 ]
print(x2) # Wypisze: [256, 64, 16, 4, 1, 0]
```

W pierwszym wierszu deklarujemy prosta tablicę x1 zawierającą dziewięć wartości, w drugim tworzymy nową listę x2 złożoną z wartości będących kwadratami wartości pobranych z listy x1, ale tylko tych, które są mniejsze lub równe 16 (zakres zaznaczyłem pogrubioną czcionką).

Możesz spróbować wymyślić wiele podobnych konstrukcji; np. szybkie utworzenie 50-elementowej listy zawierającej wszędzie liczbę 5 może wyglądać tak:

```
x3 = [ 5 for x in range(50) ].
```

Zbiory

Zbiory w Pythonie są realizacją pojęcia znanego z lekcji matematyki i dość dobrze odzwierciedlają zarówno koncepcję znaną z nauk matematycznych, jak i nasze intuicyjne jej rozumienie. W poprzednim rozdziale pokazałem przykładową realizację nieco zawężonej wersji tego typu danych, teraz przyjrzyjmy się, jak została ona zaprojektowana w samym Pythonie jako klasa standardowa.

W tabeli 8.4 podsumowałem charakterystyczne cechy zbiorów w Pythonie.

Na zbiorach można dokonywać klasycznych, matematycznych operacji: suma, przecięcie i różnica, co pokażę nieco dalej.

Na razie przypatrz się kilku prostym operacjom, aby wyczuć „filozofię", na której opiera się ten typ danych [*zbiory.py* (fragment)]:

```
zbiornik = {"żaba", "komar", "insekt"}
print ("Zbiór: ", zbiornik)
print ("Spróbujmy dodać do zbiornika kolejną żabę...")
zbiornik.add("żaba")
print ("Aktualny stan zbiornika:")
print ("Zbiór: ", zbiornik)

listal=[2,3,4,5,6,6,6,9,9,0]
```