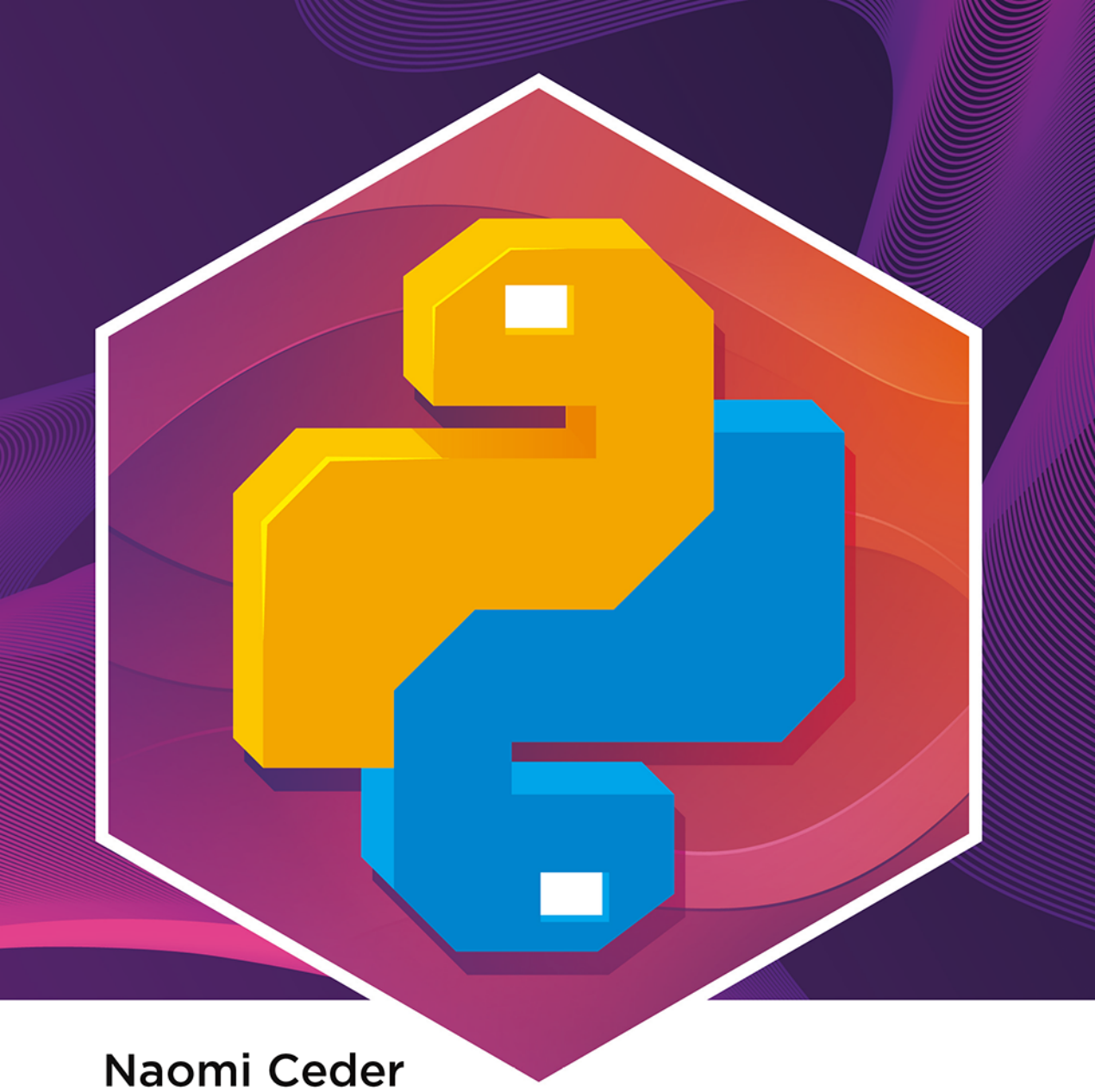


Naomi Ceder

PYTHON

SZYBKO I PROSTO

Wydanie III

Helion

Tytuł oryginału: The Quick Python Book, 3rd Edition

Tłumaczenie: Katarzyna Bogusławska

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-283-3771-8

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pyszy3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

5 Listy, krotki i zbiory

Ten rozdział zawiera omówienie:

- działań na listach i indeksach list,
- modyfikacji list,
- sortowania,
- zastosowania typowych działań na listach,
- obsługi zagnieżdżonych list i głębokich kopii,
- korzystania z krotek,
- tworzenia i wykorzystywania zbiorów.

W tym rozdziale zajmiemy się dwoma podstawowymi typami sekwencji w Pythonie — listami i krotkami. Pozornie listy mogą przypominać tablice znane z innych języków programowania, ale to tylko pierwsze wrażenie — listy są zdecydowanie bardziej elastyczne i przedstawiają znacznie więcej możliwości niż zwykłe tablice.

Krotki to listy, które nie mogą być modyfikowane. Można o nich myśleć, jak o listach z ograniczeniami czy prostym typie rekordowym. Potrzebę istnienia takiej ograniczonej struktury danych zarysuję w dalszej części rozdziału. Na kolejnych stronach przejdziemy także do nowszego typu kolekcji w Pythonie — zbioru. Zbiory sprawdzają się, gdy kładziemy nacisk na zawieranie się elementu w pewnym ciągu, a nie pozycję tego obiektu. Większa część tego rozdziału poświęcona jest listom, ponieważ zrozumienie sposobu funkcjonowania list pozwala na opanowanie wiedzy dotyczącej krotek. Przy końcu rozdziału nakreślimy natomiast różnice między listami a krotkami pod względem funkcjonalności i struktury.

5.1. Listy a tablice

Lista dla Pythona jest pod wieloma względami tym samym, czym tablica jest dla Javy, C czy jakiegokolwiek innego języka programowania. Jest to uporządkowana kolekcja obiektów. Tworzymy ją, zamykając w nawiasach kwadratowych ciągi oddzielonych przecinkami elementów, w ten sposób:

```
# poniższy kod przypisuje trójelementową listę do x
>>> x = [1, 2, 3]
```

Zwróć uwagę, że nie trzeba martwić się deklarowaniem listy bądź określaniem jej rozmiaru. W powyższym przykładzie tworzymy i przypisujemy listę, która rośnie i kurczy się w miarę potrzeb.

Tablice w Pythonie

Dostępny w Pythonie moduł `array` daje możliwość pracy z typizowanymi tablicami opartymi na strukturach danych języka C. Więcej informacji o tym module dostępne jest w dokumentacji *Python Library Reference*, ale korzystanie z niego zaleca się tylko w przypadkach, gdy kluczowe są zaawansowane optymalizacje wykorzystania zasobów. Jeśli z kolei system wymaga złożonych obliczeń matematycznych, warto sięgnąć po pakiet `NumPy`, opisany w rozdziale 4. i dostępny pod adresem *www.scipy.org*.

W przeciwieństwie do ograniczeń w wielu innych językach, listy w Pythonie mogą zawierać różne typy obiektów. Element listy może być dowolnym obiektem tego języka. Oto przykład listy zawierającej wiele elementów:

```
# pierwszy element listy to liczba, drugi — łańcuch znaków, trzeci — inna lista
>>> x = [2, "dwa", [1, 2, 3]]
```

Prawdopodobnie najprostszą wbudowaną funkcją operującą na listach jest funkcja `len`, która zwraca liczbę elementów sekwencji:

```
>>> x = [2, "dwa", [1, 2, 3]]
>>> len(x)
3
```

Zwróć uwagę, że funkcja `len` nie zlicza elementów wewnętrznej, zagnieżdżonej listy.

SZYBKI TEST: LEN Co zwróciłaby funkcja `len` wywołana dla każdej z list poniżej:

`[0]`; `[]`; `[[1, 3, [4, 5], 6], 7]`?

5.2. Indeksy list

Zrozumienie tego, jak funkcjonują indeksy list, otworzy przed Tobą wiele możliwości w Pythonie, więc poświęć nieco uwagi na przeczytanie kolejnych stron.

Do elementów listy w Pythonie można odwoływać się przy użyciu tej samej notacji, co w języku C. Zarówno w Pythonie, C, jak i wielu innych językach indeksowanie listy zaczyna się od 0 — zatem odwołanie się do elementu o indeksie 0 zwróci pierwszy element na liście, odwołanie do elementu o indeksie 1 — drugi, itd. Oto kilka przykładów:

```
>>> x = ["pierwszy", "drugi", "trzeci", "czwarty"]
>>> x[0]
'pierwszy'
>>> x[2]
'trzeci'
```

Korzystanie z indeksów w Pythonie daje jednak więcej możliwości niż w C. Jeśli indeksy mają wartości ujemne, oznaczają pozycje liczone od końca listy. W ten sposób indeks *–1* odnosi się do ostatniego elementu listy, *–2* — przedostatniego itd. Mając w pamięci stworzoną wcześniej listę `x`, można bez błędu wykonać poniższe przypisania:

```
>>> a = x[-1]
>>> a
'czwarty'
>>> a = x[-2]
>>> a
'trzeci'
```

Tam, gdzie wykorzystujemy tylko jeden indeks, można śmiało wyobrażać sobie go jako wskazanie konkretnego elementu listy. Dla bardziej skomplikowanych instrukcji poprawniej będzie myśleć o indeksie jako o wskazaniu pozycji *pomiędzy* elementami.

x=[		"pierwszy",		"drugi",		"trzeci",		"czwarty"		]
Indeks wyrażony liczbą dodatnią	0		1		2		3			
Indeks wyrażony liczbą ujemną	–4		–3		–2		–1			

Nie ma to znaczenia, gdy odnosimy się do pojedynczego elementu sekwencji, ale staje się istotne w świetle możliwości wycinania list (ang. *slicing*), czyli mechanizmu odwoływania się do całego podzbioru danej listy. Zamiast pojedynczo przypisywać każdy element poprzez notację *lista*`[`*indeks*`]`, możemy skorzystać ze składni *lista*`[`*indeks_początkowy*`:` *indeks_końcowy*`]`, by uzyskać podzbiór obejmujący wszystkie elementy, od tego o indeksie początkowym (włącznie) aż po indeks końcowy (wyłącznie). Poniżej kilka przykładów:

```
>>> x = ["pierwszy", "drugi", "trzeci", "czwarty"]
>>> x[1:-1]
['drugi', 'trzeci']
>>> x[0:3]
['pierwszy', 'drugi', 'trzeci']
>>> x[-2:-1]
['trzeci']
```

Mogłoby wydawać się intuicyjne, byśmy — jeśli indeks początkowy jest liczbą większą od indeksu końcowego — otrzymali ciąg elementów w odwróconej kolejności. Tak się jednak nie stanie. Taki kod zwróci pustą listę:

```
>>> x[-1:2]
[]
```

Kiedy wycinamy podzbiór z listy, możemy pominąć podawanie indeksu początkowego lub indeksu końcowego. Nie podając indeksu początkowego, każemy interpreterowi zacząć od początku listy, natomiast nie podając indeksu końcowego, nakazujemy kontynuować do jej końca.

```
>>> x[:3]
['pierwszy', 'drugi', 'trzeci']
>>> x[2:]
['trzeci', 'czwarty']
```

Pominięcie zarówno indeksu początkowego, jak i końcowego skutkuje stworzeniem nowej listy, zawierającej wszystkie elementy od początku do końca oryginalnej listy — dokładniej rzecz biorąc, skopiowaniem pierwotnej listy. To rozwiązanie jest przydatne, gdy potrzebujemy kopii, którą możemy modyfikować bez ryzyka modyfikacji danych w pierwotnej sekwencji.

```
>>> y = x[:]
>>> y[0] = "1."
>>> y
['1.', 'drugi', 'trzeci', 'czwarty']
>>> x
['pierwszy', 'drugi', 'trzeci', 'czwarty']
```

WYPRÓBUJ: WYCINANIE LIST I INDEKSY Biorąc pod uwagę to, co już wiesz o funkcji `len` i wycinaniu list, jak skorzystałbyś z tych dwóch narzędzi, by otrzymać drugą połowę listy o nieznanej długości? Poćwicz w konsoli Pythona, by upewnić się, że Twoje rozwiązania działają.

5.3. Modyfikowanie list

Możemy posługiwać się indeksami nie tylko po to, by uzyskiwać dostęp do elementów sekwencji, ale także po to, by te elementy zmieniać. Indeks należy wtedy umieścić po lewej stronie przypisania:

```
>>> x = [1, 2, 3, 4]
>>> x[1] = "dwa"
>>> x
[1, 'dwa', 3, 4]
```

Mechanizm wycinania list również może zostać użyty w tym miejscu. Konstrukcja typu *`lista_a[indeks1:indeks2] = lista_b`* spowoduje, że wszystkie elementy o indeksach pomiędzy *`indeksem1`* a *`indeksem2`* zostaną zastąpione elementami `listy_b`. `lista_b` może mieć mniej lub więcej elementów niż określony indeksami wycinek *`listy_a`* i w takiej sytuacji długość *`listy_a`* ulegnie zmianie. Wycinania w połączeniu z przypisaniem można używać w wielu przypadkach. Spójrzmy:

```
>>> x = [1, 2, 3, 4]
>>> x[len(x):] = [5, 6, 7]  <-- Doklejenie kolejnej listy na końcu listy
>>> x
```

```
[1, 2, 3, 4, 5, 6, 7]
>>> x[:0] = [-1, 0]  <-- Doklejenie kolejnej listy na początku listy
>>> x
[-1, 0, 1, 2, 3, 4, 5, 6, 7]
>>> x[1: -1] = []    <-- Usunięcie elementów z listy
>>> x
[-1, 7]
```

Dodawanie pojedynczego elementu do listy jest na tyle powszechną operacją, że istnieje do jej wykonania specjalna metoda `append`:

```
>>> x = [1, 2, 3]
>>> x.append("cztery")
>>> x
[1, 2, 3, 'cztery']
```

Problem może pojawić się przy dodawaniu jednej listy do drugiej. Ta druga lista zostanie dołączona na koniec sekwencji jako pojedynczy element:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.append(y)
>>> x
[1, 2, 3, 4, [5, 6, 7]]
```

Metoda `extend` jest podobna do `append`, z tą różnicą, że `extend` pozwala scalić elementy dwóch list:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 5, 6, 7]
```

Mamy także do dyspozycji metodę `insert`, która umożliwia wstawienie elementu pomiędzy dwa istniejące już elementy lub na początku listy. `insert` wywoływana jest jako metoda obiektu listy i przyjmuje dwa argumenty. Pierwszym z nich jest indeks pozycji, pod którą wstawiony ma zostać nowy element, a drugim — sam ten element.

```
>>> x = [1, 2, 3]
>>> x.insert(2, "witaj")
>>> print(x)
[1, 2, 'witaj', 3]
>>> x.insert(0, "start")
>>> print(x)
['start', 1, 2, 'witaj', 3]
```

Metoda `insert` liczy indeksy tak, jak było to opisane wcześniej w podrozdziale 5.2 (czyli zaczynając od 0), ale wydaje się, że łatwiej myśleć o zapisie `insert(n, elem)` jako oznaczającym *wstaw elem tuż przed n-tym elementem sekwencji*. Te same możliwości co `insert` przedstawia także wycinanie, a zatem `lista.insert(n, elem)` jest równoznaczne z `lista[n:n] = [elem]`, gdy `n` jest liczbą dodatnią. Używanie funkcji `insert` sprzyja jednak czytelności kodu i pozwala poradzić sobie z ujemnymi indeksami:

```
>>> x = [1, 2, 3]
>>> x.insert(-1, "witaj")
>>> print(x)
[1, 2, 'witaj', 3]
```

Instrukcja `del` jest preferowanym przez twórców języka sposobem usuwania elementów listy bądź jej wycinków. Nie robi ona nic ponad to, co można osiągnąć dzięki wycinaniu, ale zwykle jest łatwiejsza do zapamiętania i przeczytania.

```
>>> x = ['a', 2, 'c', 7, 9, 11]
>>> del x[1]
>>> x
['a', 'c', 7, 9, 11]
>>> del x[:2]
>>> x
[7, 9, 11]
```

Podsumowując, `del lista[n]` zrobi to samo, co `lista[n:n+1] = []`, podczas gdy `del lista[m:n]` wykona to samo, co `lista[m:n] = []`.

Metoda `remove` nie jest bliźniakiem `insert`. O ile `insert` wstawi obiekt w oznaczone miejsce, `remove` poszuka pierwszego wystąpienia tego obiektu i usunie je:

```
>>> x = [1, 2, 3, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 5]
>>> x.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Jeśli `remove` nie znajdzie nic do usunięcia, zgłosi błąd. Możemy starać się złapać ten błąd przy użyciu mechanizmów obsługi błędów w Pythonie lub uniknąć jego zgłoszenia, sprawdzając przy pomocy operatora `in`, czy element znajduje się na liście, zanim spróbujemy go usunąć.

Metoda `reverse` jest nieco bardziej szczegółowym sposobem modyfikowania listy. Sprawnie pod względem wydajności odwraca kolejność elementów listy w miejscu.

```
>>> x = [1, 3, 5, 6, 7]
>>> x.reverse()
>>> x
[7, 6, 5, 3, 1]
```

WYPRÓBUJ: MODYFIKOWANIE LIST Zakładając, że masz listę zawierającą 10 elementów, w jaki sposób przeniesiesz jej ostatnie trzy elementy z końca na początek, zachowując ich porządek?

5.4. Sortowanie list

Listy można sortować przy użyciu standardowej dla Pythona metody `sort`:

```
>>> x = [3, 8, 4, 0, 2, 1]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4, 8]
```

Metoda `sort` operuje na liście w miejscu, czyli zmienia oryginalną listę. By posortować wartości na liście, nie zmieniając przy tym oryginału, można wybrać jedno z dwóch rozwiązań. Można użyć funkcji `sorted`, opisanej w punkcie 5.4.2, lub skopiować listę i na tej kopii wywołać metodę `sort`:

```
>>> x = [2, 4, 1, 3]
>>> y = x[:]
>>> y.sort()
>>> x
[2, 4, 1, 3]
```

Sortowanie sprawdzi się także dla łańcuchów znaków:

```
>>> x = ["życie", "jest", "cudowne"]
>>> x.sort()
>>> x
['cudowne', 'jest', 'życie']
```

Metoda `sort` może posortować niemal wszystko, bo Python jest w stanie niemal wszystko porównać ze sobą. Niemniej i tu znajdą się ograniczenia. Domyślny sposób sortowania używany przez metodę `sort` wymaga, by wszystkie elementy listy były typami dającymi się porównać. Z tego powodu, próbując sortować listę zawierającą zarówno liczby całkowite, jak i łańcuchy znaków, interpreter zgłosi błąd:

```
>>> x = [1, 2, "witaj", 3]
>>> x.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

Da się jednak posortować w ten sposób listę list:

```
>>> x = [[3, 5], [2, 9], [2,3], [4, 1], [3, 2]]
>>> x.sort()
>>> x
[[2, 3], [2, 9], [3, 2], [3, 5], [4, 1]]
```

Zgodnie ze standardowymi algorytmami porównywania złożonych obiektów w Pythonie, elementy będące listami są najpierw sortowane rosnąco względem swojego pierwszego elementu, następnie rosnąco względem drugiego elementu.

Metoda `sort` jest jednak jeszcze potężniejszym narzędziem. Przyjmuje opcjonalny parametr `reverse`, który, jeśli zostanie ustawiony na `True`, posortuje listę malejąco. Co więcej, można używać własnej funkcji, która określi, jak elementy będą posortowane.

5.4.1. Własne mechanizmy sortowania

By korzystać z własnych mechanizmów sortowania, musimy być w stanie definiować funkcje — a o tym jeszcze nie było mowy. W następnych akapitach opiszemy też fakt, że

funkcja `len(łańcuch_znaków)` zwraca liczbę znaków w łańcuchu. Jednak pełniej o łańcuchach znaków będziemy mówić dopiero w rozdziale 6.

Domyślnie Python korzysta z własnych wbudowanych funkcji określających uporządkowanie zbioru, co wystarcza w większości zastosowań. Zdarza się jednak czasem potrzeba posortowania elementów listy w sposób inny niż domyślne zachowanie interpretera. Wyobraźmy sobie konieczność posortowania listy łańcuchów znaków pod względem liczby znaków w słowie, a nie alfabetycznie (co byłoby standardowym zachowaniem języka).

By osiągnąć ten cel, potrzebujemy funkcji, która zwróci klucz, z którego `sort` skorzysta do uporządkowania elementów. Do porządkowania wyrazów względem liczby znaków odpowiednią funkcją zwracającą klucz byłoby:

```
>>> def porownaj_liczbe_znakow((wyraz1):
...     return len(wyraz1)
```

Powyższa funkcja jest bardzo prosta. Przekazuje ona do metody `sort` długość łańcucha znaków.

Dysponując taką funkcją, możemy przekazać ją do metody `sort` przy użyciu słowa kluczowego `key`. W związku z tym, że funkcje w Pytonie są obiektami, możemy przekazywać je jako argumenty funkcji, tak jak dowolny inny typ obiektu. Oto krótki program obrazujący różnicę pomiędzy domyślnym sortowaniem a własnym mechanizmem sortowania:

```
>>> def porownaj_liczbe_znakow((wyraz1):
...     return len(wyraz1)
...
>>> lista_wyrazow = ["Python", "jest", "lepszy", "od", "C"]
>>> lista_wyrazow.sort()
>>> print(lista_wyrazow)
['C', 'Python', 'jest', 'lepszy', 'od']
>>> lista_wyrazow = ["Python", "jest", "lepszy", "od", "C"]
>>> lista_wyrazow.sort(key=porownaj_liczbe_znakow()
>>> print(lista_wyrazow)
['C', 'od', 'jest', 'Python', 'lepszy']
```

Pierwsza lista uporządkowana jest alfabetycznie (a w takim wypadku wielkie litery występują przed małymi), a w drugiej kolejność wyznacza rosnąca liczba znaków w słowie.

Własne mechanizmy sortowania są użyteczne, ale pod względem wydajności mogą wypadać gorzej niż domyślne algorytmy. Zwykle straty są niewielkie, ale jeśli funkcja sortująca jest bardzo złożona, to jej wydajność może być niska dla list o setkach tysięcy czy milionach elementów.

Miejscem, w którym należy szczególnie wystrzegać się własnych mechanizmów sortowania, jest porządkowanie listy w kolejności malejącej. W takich sytuacjach zaleca się używanie metody `sort` z parametrem `reverse` ustawionym na `True`. Jeśli z jakichś powodów nie chcemy tego robić, nadal — pod względem wydajności — lepsze będzie posortowanie tej listy przy użyciu `sort`, a następnie skorzystanie z metody `reverse`, by odwrócić porządek nowej listy. Obie te operacje razem — sortowanie rosnące i odwrócenie kolejności — będą o wiele szybsze niż własny algorytm sortujący.

5.4.2. Funkcja sorted

Listy mają wbudowaną metodę do sortowania samych siebie, ale inne iterowalne obiekty w Pythonie, jak na przykład klucze słowników, takich możliwości nie posiadają. Python ma ponadto wbudowaną funkcję `sorted`, która zwraca posortowaną sekwencję z dowolnego iterowalnego obiektu. Przyjmuje ona takie same parametry `key` oraz `reverse`, co `sort`:

```
>>> x = [4, 3, 1, 2]
>>> y = sorted(x)
>>> y
[1, 2, 3, 4]
>>> z = sorted(x, reverse=True)
>>> z
[4, 3, 2, 1]
```

WYPRÓBUJ: SORTOWANIE LIST Wyobraźmy sobie, że mamy do czynienia z listą, której każdy element również jest listą: `[[1, 2, 3], [2, 1, 3], [4, 0, 1]]`. Gdybyśmy chcieli posortować taką listę pod względem wartości drugiego elementu każdej z list tak, by otrzymać wynik w postaci `[[4, 0, 1], [2, 1, 3], [1, 2, 3]]`, z jakiej własnej funkcji sortującej powinniśmy skorzystać i przekazać ją jako parametr `key` w wywołaniu `sort`?

5.5. Inne przydatne działania na listach

Kilka innych metod list jest często używanych, ale nie pasuje do żadnej określonej kategorii.

5.5.1. Przynależność do zbioru i operator in

Łatwo sprawdzić, czy wartość znajduje się na liście, dzięki operatorowi `in`, który zwraca wartość logiczną prawda/fałsz. Można także używać negacji `not in`:

```
>>> 3 in [1, 3, 4, 5]
True
>>> 3 not in [1, 3, 4, 5]
False
>>> 3 in ["jeden", "dwa", "trzy"]
False
>>> 3 not in ["jeden", "dwa", "trzy"]
True
```

5.5.2. Konkatenacja list i operator +

By stworzyć listę z połączenia dwóch istniejących list, powinniśmy użyć operatora konkatenacji list +, który pozostawia oryginalne listy nietknięte:

```
>>> z = [1, 2, 3] + [4, 5]
>>> z
[1, 2, 3, 4, 5]
```

5.5.3. Inicjalizacja listy i operator *

Z operatora * korzystamy, chcąc stworzyć listę o określonej wielkości zainicjalizowaną daną wartością. Operacja ta jest często stosowana przy pracy z olbrzymimi listami, których wielkość jest od początku znana. Chociaż można używać funkcji `add`, by dodawać

elementy do listy i w ten sposób ją powiększać, lepsze wyniki osiąga się, poprawnie inicjalizując listę na początku programu. Lista, której wielkość nie zmienia się w trakcie pracy, nie powoduje narzutu związanego z relokacją pamięci.

```
>>> z = [None] * 4
>>> z
[None, None, None, None]
```

Gdy używamy * w takim kontekście, symbol ten nazywany jest operatorem mnożenia list i powiela wskazaną listę oznaczoną liczbę razy, a następnie łączy wszystkie kopie w jedną, nową listę. Jest to przyjęta metoda tworzenia list o zdefiniowanej wielkości w Pythonie. Lista zawierająca pojedynczy element `None` jest powszechnie używana w tym celu, ale typ elementu inicjalizującego może być dowolny:

```
>>> z = [3, 1] * 2
>>> z
[3, 1, 3, 1]
```

5.5.4. Maksymalna i minimalna wartość elementu oraz funkcje max i min

Twórcy języka wyposażyli nas w funkcje `min` i `max` w celu znalezienia najmniejszego i największego elementu listy. Najczęściej stosuje się je z listami zawierającymi elementy liczbowe, ale można ich używać w odniesieniu do list dowolnych typów. Próba znalezienia wartości minimalnej i maksymalnej w sekwencji obiektów różnych typów kończy się błędem, jeśli porównanie tych typów nie znajdzie wspólnej podstawy:

```
>>> min([3, 7, 0, -2, 11])
-2
>>> max([4, "Witaj", [1, 2]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'int'
```

5.5.5. Przeszukiwanie listy i metoda index

Jeśli mamy za zadanie znaleźć miejsce, w którym dana wartość występuje (a nie tylko dowiedzieć się, czy w ogóle występuje), możemy posłużyć się metodą `index`. Metoda ta przeszukuje listę w poszukiwaniu elementu odpowiadającego zadanej wartości i zwraca jego indeks:

```
>>> x = [1, 2, "pięć", 7, -2]
>>> x.index(7)
3
>>> x.index(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in list
```

Jak widać w przykładzie wyżej, odpytywanie o pozycję elementu, którego brak na liście, skutkuje błędem. Może on zostać obsłużony w sposób analogiczny do tego zgłaszanego przez funkcję `remove` (czyli poprzez sprawdzenie przy użyciu `in` przed wywołaniem `index`).

5.5.6. Wystąpienia elementu i metoda count

Metoda `count` również przeszukuje listę w poszukiwaniu zadanej wartości, ale zamiast zwracać jej indeks, podaje liczbę wystąpień tej wartości na liście.

```
>>> x = [1, 2, 2, 3, 5, 2, 5]
>>> x.count(2)
3
>>> x.count(5)
2
>>> x.count(4)
0
```

5.5.7. Podsumowanie działań na listach

Jak widać, listy to przydatne struktury danych, o możliwościach wykraczających daleko ponad to, co było możliwe ze starymi, dobrymi tablicami. Operacje na listach są niezwykle istotne w programowaniu w Pythonie, dlatego warto podsumować je w obrazowym zestawieniu w tabeli 5.1.

Tabela 5.1. Działania na listach

Działanie	Opis	Przykład
`[]`	Tworzy pustą listę	`x = []`
`len`	Zwraca długość listy	`len(x)`
`append`	Dodaje pojedynczy element na koniec listy	`x.append('y')`
`extend`	Dodaje listę na koniec bieżącej listy	`x.extend(['a', 'b'])`
`insert`	Wstawia nowy element w oznaczone miejsce listy	`x.insert(0, 'y')`
`del`	Usuwa element lub wycinek listy	`del(x[0])`
`remove`	Wyszukuje i usuwa oznaczony element listy	`x.remove('y')`
`reverse`	Odwraca kolejność listy w miejscu	`x.reverse()`
`sort`	Sortuje listę w miejscu	`x.sort()`
`+`	Scala dwie listy	`x1 + x2`
`*`	Powiela listę	`x = ['y'] * 3`
`min`	Zwraca minimalną wartość na liście	`min(x)`
`max`	Zwraca maksymalną wartość na liście	`max(x)`
`index`	Zwraca indeks elementu na liście	`x.index('y')`
`count`	Liczy liczbę wystąpień elementu na liście	`x.count('y')`
`sum`	Sumuje elementy (o ile mogą zostać dodane)	`sum(x)`
`in`	Zwraca informację o tym, czy element znajduje się na liście	`'y' in x`

Biegłość w posługiwaniu się wymienionymi wyżej operacjami na listach znacząco ułatwi życie każdemu programiście Pythona.

SZYBKI TEST: DZIAŁANIA NA LISTACH Jaki wynik zwróci `len([[1,2]] * 3)`?
Wskaż dwie różnice między operatorem `in` a metodą `index`.
Które wywołanie zgłosi wyjątek?

```
min(["a", "b", "c"])
max([1, 2, "trzy"])
[1, 2, 3].count("jeden")
```

WYPRÓBUJ: DZIAŁANIA NA LISTACH Napisz kod bezpiecznie usuwający element z listy `x` wtedy i tylko wtedy, gdy element ten znajduje się na liście `x`.

Zmień przygotowany kod tak, by usuwał element z listy, jeśli element ten występuje na liście więcej niż raz.

5.6. Listy zagnieżdżone i kopie głębokie

W tym podrozdziale prezentowany jest temat nieco bardziej zaawansowany, który możesz pominąć, jeśli dopiero rozpoczynasz naukę Pythona.

Listy mogą być zagnieżdżone. Jednym z zastosowań takiej konstrukcji może być reprezentowanie macierzy dwuwymiarowej. Do elementów takich macierzy można odwoływać się przy użyciu indeksów dwuwymiarowych. Z indeksów tych korzystamy w następujący sposób:

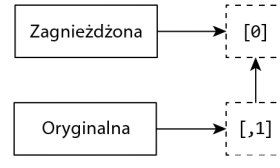
```
>>> m = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
>>> m[0]
[0, 1, 2]
>>> m[0][1]
1
>>> m[2]
[20, 21, 22]
>>> m[2][2]
22
```

Mechanizm ten oczywiście rozszerza się do obsługi macierzy wielowymiarowych w sposób łatwy do przewidzenia.

Najczęściej na tym można poprzestać, jeśli chodzi o posługiwanie się listami zagnieżdżonymi. Czasami jednak można napotkać pewne szczególne problemy z tymi strukturami danych, zwłaszcza jeśli chodzi o odwoływanie się poprzez zmienną do obiektu oraz sposób, w jaki pewne obiekty (np. listy) mogą być modyfikowane (są mutowalne). Najłatwiej będzie zobrazować to przykładem:

```
>>> zagnieżdżona = [0]
>>> oryginalna = [zagnieżdżona, 1]
>>> oryginalna
[[0], 1]
```

Rysunek 5.1 obrazuje ten przykład.



Rysunek 5.1. Lista z pierwszym elementem referującym do zagnieżdżonej listy

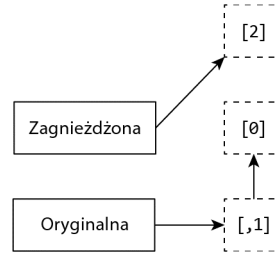
Wartość w zagnieżdżonej liście może zostać zmieniona na dwa sposoby — odnosząc się do zmiennej w liście `zagnieżdżona` bądź w liście `oryginalna`.

```
>>> zagnieżdżona[0] = 'zero'
>>> oryginalna
[['zero'], 1]
>>> oryginalna[0][0] = 0
>>> zagnieżdżona
[0]
>>> oryginalna
[[0], 1]
```

Jednakże jeśli wartość listy `zagnieżdżona` zostanie ustawiona na inną listę, więź między listami `zagnieżdżona` i `oryginalna` zostanie zerwana.

```
>>> zagnieżdżona = [2]
>>> oryginalna
[[0], 1]
```

Rysunek 5.2 ilustruje ten warunek.

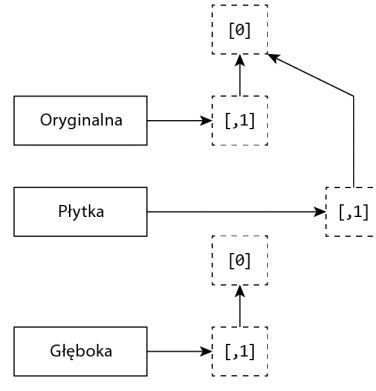


Rysunek 5.2. Pierwszy element listy oryginalnej nadal jest zagnieżdżony, ale zmienna zagnieżdżona wskazuje na inną listę

Widzieliśmy, że możemy skopiować listę, biorąc z niej pełny wycinek, czyli stosując składnię `x[:]`. Możemy także otrzymać kopię listy, używając operatora `+` lub `*` (np. `x + []` lub `x * 1`). Te dwie metody wypadają nieco słabiej pod względem wydajności niż wycinanie. Jednak wszystkie trzy tworzą strukturę nazywaną płytką kopią — i najczęściej właśnie takiej kopii potrzebujemy. Jeśli jednak kopiowana lista zawiera zagnieżdżone listy, możemy potrzebować tzw. kopii głębokiej. Możemy ją uzyskać, korzystając z funkcji kopiowania głębokiego w module `copy`:

```
>>> oryginalna = [[0], 1]
>>> płytka = oryginalna[:]
>>> import copy
>>> głęboka = copy.deepcopy(oryginalna)
```

Sięgnijmy do rysunku 5.3 celem ilustracji.



Rysunek 5.3. Płytka kopia nie kopiuje zagnieżdżonych list

Listy, na które wskazują zmienna `oryginalna` oraz `płytka`, są połączone. Zmiana wartości na zagnieżdżonej liście z poziomu którejkolwiek z nich poskutkuje zmianami dla obu.

```
>>> płytka[1] = 2
>>> płytka
[[0], 2]
>>> oryginalna
[[0], 1]
>>> płytka[0][0] = 'zero'
>>> oryginalna
[['zero'], 1]
```

Kopia głęboka jest niezależna od oryginału i żadna zmiana dokonywana na kopii nie wpływa na oryginał.

```
>>> głęboka[0][0] = 5
>>> głęboka
[[5], 1]
>>> oryginalna
[['zero'], 1]
```

Zachowanie to jest analogiczne dla wszystkich innych zagnieżdżonych obiektów, które można modyfikować (np. słowników).

A teraz, gdy widzieliśmy już, ile potrafią listy, przejdźmy do krotek.

WYPRÓBUJ: KOPIOWANIE LIST Mając poniższą listę:

```
x = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
```

jaki kod należałoby przygotować, by uzyskać taką kopię, na której zmiany nie będą wpływały na zawartość listy `x`?

5.7. Krotki

Krotki to struktury danych bardzo podobne do list, ale niepodlegające modyfikacji po utworzeniu. Wyjąwszy tę różnicę, są na tyle podobne do list, że można by zastanawiać się, po co twórcy języka w ogóle je udostępnili. Powodem jest to, że krotki w efektywny sposób pełnią funkcje, których listy nie mogą pełnić — jak na przykład stanowienie kluczy w słownikach.

5.7.1. Podstawy krotek

Tworzenie krotki jest podobne do tworzenia listy i sprowadza się do przypisania ciągu wartości do zmiennej. Lista to sekwencja ujęta w nawiasy kwadratowe — `[]` — a krotka to z kolei sekwencja ujęta w nawiasy okrągłe — `()`:

```
>>> x = ('a', 'b', 'c')
```

Powyższa linia tworzy trójelementową krotkę.

Po utworzeniu krotka jest tak podobna do listy, że łatwo zapomnieć, że to dwa różne typy danych:

```
>>> x[2]
'c'
>>> x[1:]
('b', 'c')
>>> len(x)
3
>>> max(x)
'c'
>>> min(x)
'a'
>>> 5 in x
False
>>> 5 not in x
True
```

Podstawowa różnica między listami a krotkami polega na tym, że krotki nie są mutowalne. Próba zmiany krotki skutkuje wyświetleniem wprowadzającego dezorientację komunikatu, który jest w Pythonie sposobem zakomunikowania, że interpreter nie wie, jak ustawić wartość elementu w krotce:

```
>>> x[2] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Można jednak tworzyć nowe krotki na podstawie już istniejących przy użyciu operatorów + oraz *:

```
>>> x + x
('a', 'b', 'c', 'a', 'b', 'c')
>>> 2 * x
('a', 'b', 'c', 'a', 'b', 'c')
```

Kopię krotki możemy wykonać na wszystkie sposoby, które dostępne są dla powielania list:

```
>>> x[:]
('a', 'b', 'c')
>>> x * 1
('a', 'b', 'c')
>>> x + ()
('a', 'b', 'c')
```

Jeśli nie czytałeś podrozdziału 5.6, możesz pominąć resztę tego akapitu. Krotki same w sobie nie mogą być zmieniane. Ale jeśli składają się z mutowalnych obiektów (np. list lub słowników), obiekty te mogą ulec zmianie, o ile ich elementy przypisane są do własnych zmiennych. Krotki, które zawierają mutowalne obiekty, nie mogą być używane jako klucze w słownikach.

5.7.2. Jednoelementowe krotki wymagają przecinka

Przy używaniu krotek należy pamiętać o istotnej uwadze dotyczącej składni. Ponieważ nawiasy kwadratowe, w które ujmuje się listy, nie są w żaden inny sposób używane w Pythonie, jasne jest, że zapis `[]` oznacza pustą listę, a `[1]` oznacza listę z jednym elementem. Jednak nie można tego samego powiedzieć o nawiasach używanych z krotkami. Nawiasy okrągłe mogą być także stosowane do grupowania elementów w wyrażeniach, by wymusić pewną kolejność ich interpretowania. Z tego powodu nie jest oczywiste, czy fragment kodu `(x + y)` w programie napisanym w Pythonie należy rozumieć jako polecenia dodania `x` oraz `y` i umieszczenia sumy w jednoelementowej krotce, czy też może polecenie dodania `x` i `y`, zanim wykona się jakiekolwiek inne wyrażenie sąsiadujące z nim.

Taka niejasność zachodzi tylko dla jednoelementowych krotek, ponieważ krotki o większej liczbie elementów zawsze zawierają przecinki, które oddzielają elementy i wskazują, że oddzielone nimi wartości znajdują się w krotce. Aby uniknąć tej niejasności, Python wymaga, by po elemencie w jednoelementowej krotce umieścić przecinek. W przypadku pustej krotki nie mamy tego problemu — musi to być krotka, ponieważ w innym wypadku para okrągłych nawiasów pozbawiona byłaby znaczenia:

```
>>> x = 3
>>> y = 4
>>> (x + y)  # Ta linia dodaje x do y
7
>>> (x + y,) # Dodanie przecinka wskazuje, że nawiasy okrągłe oznaczają krotkę
(7,)
>>> ()       # By stworzyć pustą krotkę, używamy pustej pary nawiasów kwadratowych
()
```

5.7.3. Pakowanie i rozpakowywanie krotek

Jako udogodnienie Python pozwala, by krotki znajdowały się po lewej stronie operatora przypisania. W takiej składni zmiennym w krotce po lewej przypisywane są wartości z krotki po prawej stronie operatora. Oto przykład:

```
>>> (jeden, dwa, trzy, cztery) = (1, 2, 3, 4)
>>> jeden
1
>>> dwa
2
```

Powyższy przykład można zapisać jeszcze prościej, ponieważ Python rozpoznaje krotki w kontekście przypisania nawet bez okrągłych nawiasów. Wartości po prawej stronie pakowane są w krotkę, a następnie rozpakowywane do zmiennych po lewej stronie:

```
>>> jeden, dwa, trzy, cztery = 1, 2, 3, 4
```

Jedna linia kodu zastąpiła cztery:

```
>>> jeden = 1
>>> dwa = 2
>>> trzy = 3
>>> cztery = 4
```

Ta metoda jest wyjątkowo użyteczna przy podmienianiu wartości w zmiennych. Zamiast pisać:

```
temp = zmienna1
zmienna1 = zmienna2
zmienna2 = temp
```

wystarczy krótko sformułować to w postaci:

```
zmienna1, zmienna2 = zmienna2, zmienna1
```

By uprościć pracę jeszcze bardziej, Python 3 posiada rozbudowaną funkcjonalność rozpakowywania zmiennych, która pozwala elementowi oznaczonemu * pomieścić dowolną liczbę elementów niepasujących gdzie indziej. Po raz kolejny z pomocą przyjdzie przykład:

```
>>> x = (1, 2, 3, 4)
>>> a, b, *c = x
>>> a, b, c
(1, 2, [3, 4])
>>> a, *b, c = x
>>> a, b, c
(1, [2, 3], 4)
>>> *a, b, c = x
>>> a, b, c
([1, 2], 3, 4)
>>> a, b, c, d, *e = x
>>> a, b, c, d, e
(1, 2, 3, 4, [])
```

Zwróćmy uwagę, że zmienna oznaczona gwiazdką przechowuje nadwyżkę elementów jako listę, a w przypadku gdy nie ma nadwyżki elementów — przypisywana jest jej pusta lista.

Pakowanie i rozpakowywanie może być także używane z listami:

```
>>> [a, b] = [1, 2]
>>> [c, d] = 3, 4
>>> [e, f] = (5, 6)
>>> (g, h) = 7, 8
>>> i, j = [9, 10]
>>> k, l = (11, 12)
>>> a
1
>>> [b, c, d]
[2, 3, 4]
>>> (e, f, g)
(5, 6, 7)
>>> h, i, j, k, l
(8, 9, 10, 11, 12)
```

5.7.4. Konwertowanie pomiędzy listami i krotkami

Krotki można z łatwością zamienić w listy przy użyciu funkcji `list`, która przyjmuje dowolną sekwencję jako argument i zwraca listę z tymi samymi elementami, co oryginalna sekwencja. Na podobnej zasadzie listy można zamienić w krotki przy użyciu funkcji `tuple`, która robi to samo, tyle że zwraca krotkę, a nie listę.

```
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]
>>> tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

Na marginesie warto odnotować, że funkcja `list` jest dogodnym sposobem zamieniania łańcucha znaków na listę znaków:

```
>>> list("Witaj")
['W', 'i', 't', 'a', 'j']
```

Metoda ta działa, ponieważ funkcja `list` (tak jak i `tuple`) ma zastosowanie do dowolnej sekwencji w Pythonie, a łańcuch znaków jest taką właśnie sekwencją (łańcuchy znaków są szczegółowo opisane w rozdziale 6.).

SZYBKI TEST: KROTKI Wyjaśnij, dlaczego poniższe manipulacje są niedozwolone, gdy `x = (1, 2, 3, 4)`:

- `x.append(1)`,
- `x[1] = "witaj"`,
- `del x[2]`.

Gdybyśmy dysponowali krotką `x = (1, 2, 3, 4)`, jak moglibyśmy otrzymać krotkę `x` posortowaną?

5.8. Zbiory

Zbiór w Pythonie to nieuporządkowana kolekcja obiektów, używana wtedy, gdy przynależność elementu do zbioru oraz niepowtarzalność elementów w kolekcji to cechy, na których najbardziej zależy programiście. Podobnie jak klucze słowników (o których więcej w rozdziale 7.), elementy zbioru muszą być niezmienne i musi być możliwe wyliczenie z nich wartości skrótu nieodwracalnego (ang. *hash*). Znaczy to, że liczby całkowite, zmiennoprzecinkowe i krotki mogą być elementami zbioru, ale listy, słowniki i same zbiory — nie.

5.8.1. Działania na zbiorach

Oprócz operacji mających zastosowanie do kolekcji w ogólności (jak np. `in`, `len` czy iterowanie przy użyciu pętli `for`) zbiory mają kilka charakterystycznych tylko dla siebie działań:

```
>>> x = set([1, 2, 3, 1, 3, 5])  ❶
>>> x
{1, 2, 3, 5}  ❷
>>> x.add(6)  ❸
>>> x
{1, 2, 3, 5, 6}
>>> x.remove(5)  ❹
>>> x
{1, 2, 3, 6}
>>> 1 in x  ❺
True
>>> 4 in x  ❺
False
>>> y = set([1, 7, 8, 9])
>>> x | y  ❻
{1, 2, 3, 6, 7, 8, 9}
>>> x & y  ❼
{1}
>>> x ^ y  ❽
{2, 3, 6, 7, 8, 9}
```

Zbiory można tworzyć przy użyciu funkcji `set`, przyjmującej jako argument np. listę ❶. Podczas zamiany sekwencji na zbiór duplikaty są usuwane ❷. Po stworzeniu przy pomocy tej funkcji zbioru można elementy dodawać (przy użyciu funkcji `add` ❸) i usuwać (przy użyciu funkcji `remove` ❹) i w ten sposób zmieniać zbiór. Słowo kluczowe `in` jest używane do sprawdzania przynależności elementu do zbioru ❺. Można także używać operatora `|` ❻, by uzyskać sumę zbiorów, operatora `&`, by otrzymać część wspólną ❼, oraz operatora `^`, by znaleźć różnicę symetryczną zbiorów ❽ — czyli elementy występujące w jednym lub drugim zbiorze, ale nie w obu.

Powyższe elementy nie wyczerpują listy działań dozwolonych na zbiorach, ale obrazują mechanizm funkcjonowania zbiorów. Więcej informacji można uzyskać w oficjalnej dokumentacji Pythona.

5.8.2. Frozenset

Ponieważ zbiory są mutowalne i nie da się z nich wyliczyć wartości skrótu nieodwracalnego (hash), nie mogą być elementami innych zbiorów. By zaradzić tej sytuacji, Python posiada specjalny rodzaj zbiorów — frozenset (zbiór zamrożony), który działa tak, jak zwykły zbiór, z tą różnicą, że po utworzeniu nie można go już zmodyfikować. W związku z tym, że frozenset jest typem niezmiennym i pozwalającym na policzenie wartości skrótu nieodwracalnego (hash), może być elementem innego zbioru:

```
>>> x = set([1, 2, 3, 1, 3, 5])
>>> z = frozenset(x)
>>> z
frozenset({1, 2, 3, 5})
>>> z.add(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>> x.add(z)
>>> x
{1, 2, 3, 5, frozenset({1, 2, 3, 5})}
```

SZYBKI TEST: ZBIORY Gdybyś utworzył zbiór z poniższej listy, to ile miałby on elementów?

```
[1, 2, 5, 1, 0, 2, 3, 1, 1, (1, 2, 3)]
```

LABORATORIUM 5.: PRACA Z LISTAMI Podczas tego ćwiczenia mamy za zadanie wczytać z pliku zestaw odczytów temperatury (miesięczne najwyższe temperatury na lotnisku Heathrow od roku 1948 do 2016), a następnie znaleźć pewne podstawowe informacje: najwyższe i najniższe wartości pomiarów, średnią i medianę (czyli środkową wartość pomiaru, gdy wszystkie odczyty są posortowane).

Dane dotyczące odczytów temperatury znajdują się w pliku *lab_05.txt* w katalogu z kodem źródłowym przykładów z tego rozdziału. Ponieważ nie omawialiśmy jeszcze czytania plików, oto kod niezbędny do wczytania pliku i uzyskania jego zawartości w postaci listy:

```
>>> temperatury = []
>>> with open('lab_05.txt') as plik_wejściowy:
...     for wiersz in plik_wejściowy:
...         temperatury.append(int(wiersz.strip()))
```

Należy znaleźć najniższą i najwyższą temperaturę, średnią i medianę. Użyteczne w tym celu mogą być funkcje `min`, `max`, `sum`, `len` i metoda `sort`.

ZADANIE DODATKOWE Podaj liczbę odczytów, których wartość się nie powtarza.

Podsumowanie

- Listy i krotki to struktury danych oparte na zasadzie sekwencji elementów, podobnie jak łańcuchy znaków.
- Listy odpowiadają tablicom istniejącym w wielu innych językach programowania, ale posiadają funkcjonalność automatycznego dopasowywania długości, możliwość wycinania i wiele udogodnień.
- Krotki są podobne do list, ale nie mogą być modyfikowane, więc zajmują mniej pamięci i mogą być kluczami w słownikach (więcej w rozdziale 7.).
- Zbiory są iterowalnymi kolekcjami, ale są nieuporządkowane i nie mogą zawierać duplikatów.

Skorowidz

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Na początku, w 1989 roku, był tylko wakacyjnym projektem Guida van Rossuma. Dziś jest potężnym językiem programowania o niesamowitej wszechstronności: można się nim posłużyć do napisania skryptu ułatwiającego pracę administratora sieci, stworzenia aplikacji internetowej, a także opracowania systemu głębokiego uczenia maszynowego. Co więcej, dookoła Pythona skupiła się międzynarodowa społeczność tworząca niesamowite biblioteki i frameworki, co przenosi programowanie w Pythonie na zupełnie inny poziom. Ten język ma i taką ważną zaletę, że jego nauka jest przyjemna i angażująca. Nawet początkujący programista może bardzo szybko zacząć pisać poprawny i dobrze działający kod.

Dzięki tej książce zaczniesz błyskawicznie programować w Pythonie! Pominięto tu zbędne szczegóły, a skoncentrowano się na najważniejszych dla programisty, fundamentalnych zasadach programowania: przepływie sterowania, programowaniu zorientowanym obiektowo, dostępie do plików czy obsłudze wyjątków. Liczne porady, wskazówki i obszerne przykłady pomogą Ci w opanowaniu poszczególnych zagadnień. Poza omówieniem Pythona, jego najważniejszych bibliotek, pakietów i narzędzi w tym wydaniu znajdziesz pięć nowych rozdziałów dotyczących data science. Praca z tym podręcznikiem sprawi, że szybko będziesz gotów nawet na bardzo trudne zadania — i w pełni wykorzystasz potencjał Pythona!

W tej książce między innymi:

- wprowadzenie do Pythona i przygotowanie IDLE — środowiska pracy
- tworzenie kodu niezależnego od platformy
- dostęp do relacyjnych i nierelacyjnych baz danych
- obsługa wyjątków i praca na plikach
- pakiety w Pythonie

Naomi Ceder programuje w wielu językach od prawie 30 lat, pełniła też funkcję administratora systemów Linux i architekta systemów. Od 2001 roku uczy programować w Pythonie na wszystkich poziomach zaawansowania — od 12-latków po profesjonalistów. Chętnie wykłada na temat Pythona i zalet społeczności opartej na integracji. Obecnie prowadzi zespół deweloperski w Dick Blick Art Materials i jest prezesem Python Software Foundation.

Python: język elegancki, wszechstronny, elastyczny!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-3771-8

Cena: 79,00 zł