



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

Original edition copyright © 2018 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2019 by HELION SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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/pysz3>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O autorce	15
Przedmowa	17
Wprowadzenie	19
Podziękowania	21
O książce	23
CZĘŚĆ I. ZACZYAMY	27
Rozdział 1. O Pythonie	29
1.1. Czemu powinienem uczyć się właśnie Pythona?	29
1.2. W czym Python wypada dobrze?	30
1.2.1. <i>Python jest łatwy w użyciu</i>	30
1.2.2. <i>Python jest zwięzły</i>	30
1.2.3. <i>Python jest czytelny</i>	31
1.2.4. <i>Python jest kompletny</i>	32
1.2.5. <i>Python jest wieloplatformowy</i>	32
1.2.6. <i>Python jest darmowy</i>	32
1.3. Z czym Python sobie nie radzi?	33
1.3.1. <i>Python nie jest najszybszym z języków</i>	33
1.3.2. <i>Python nie ma największej liczby bibliotek</i>	34
1.3.3. <i>Python nie sprawdza typów zmiennych podczas kompilacji</i>	34
1.3.4. <i>Python słabo wspiera urządzenia mobilne</i>	34
1.3.5. <i>Python nie wykorzystuje dobrze wielu procesorów naraz</i>	34
1.4. Po co uczyć się Pythona 3?	35
Podsumowanie	36
Rozdział 2. Pierwsze kroki	37
2.1. Instalacja Pythona	37
2.2. Podstawy trybu interaktywnego i IDLE	39
2.2.1. <i>Podstawowy tryb konsolowy</i>	39
2.2.2. <i>Zintegrowane środowisko programistyczne IDLE</i>	40
2.2.3. <i>Wybór pomiędzy podstawowym trybem konsolowym a IDLE</i>	41
2.3. Używanie okna konsoli Pythona w IDLE	41
2.4. Witaj, świecie	42
2.5. Używanie konsoli do poznania możliwości Pythona	42
Podsumowanie	44

Rozdział 3. Przegląd najważniejszych zagadnień w Pythonie	45
3.1. Python w skrócie	46
3.2. Typy wbudowane	46
3.2.1. Liczby	46
3.2.2. Listy	48
3.2.3. Krotki	49
3.2.4. Łańcuchy znaków	50
3.2.5. Słowniki	51
3.2.6. Zbiory	52
3.2.7. Obiekty plików	52
3.3. Kontrola przepływu sterowania	53
3.3.1. Wartości logiczne i wyrażenia	53
3.3.2. Instrukcja if-elif-else	53
3.3.3. Pętla while	54
3.3.4. Pętla for	54
3.3.5. Definiowanie funkcji	55
3.3.6. Wyjątki	55
3.3.7. Kontekstowa obsługa błędów i słowo kluczowe with	56
3.4. Tworzenie modułów	57
3.5. Programowanie zorientowane obiektowo	58
Podsumowanie	59
CZĘŚĆ II. PODSTAWY	61
Rozdział 4. Podstawy podstaw	63
4.1. Struktura wcięć i bloków	63
4.2. Zróznicowanie komentarzy	65
4.3. Zmienne i przypisania	65
4.4. Wyrażenia	67
4.5. Łańcuchy znaków	68
4.6. Liczby	68
4.6.1. Wbudowane funkcje liczbowe	70
4.6.2. Zaawansowane funkcje liczbowe	70
4.6.3. Przeliczenia liczbowe	70
4.6.4. Liczby zespolone	70
4.6.5. Zaawansowane funkcje na liczbach zespolonych	71
4.7. Wartość None	72
4.8. Uzyskiwanie danych od użytkownika	72
4.9. Wbudowane operatory	73
4.10. Podstawy stylu typowego dla Pythona	73
Podsumowanie	73
Rozdział 5. Listy, krotki i zbiory	75
5.1. Listy a tablice	76
5.2. Indeksy list	76
5.3. Modyfikowanie list	78

5.4.	Sortowanie list	80
5.4.1.	<i>Własne mechanizmy sortowania</i>	81
5.4.2.	<i>Funkcja sorted</i>	83
5.5.	Inne przydatne działania na listach	83
5.5.1.	<i>Przynależność do zbioru i operator in</i>	83
5.5.2.	<i>Konkatenacja list i operator +</i>	83
5.5.3.	<i>Inicjalizacja listy i operator *</i>	83
5.5.4.	<i>Maksymalna i minimalna wartość elementu oraz funkcje max i min</i>	84
5.5.5.	<i>Przeszukiwanie listy i metoda index</i>	84
5.5.6.	<i>Wystąpienia elementu i metoda count</i>	85
5.5.7.	<i>Podsumowanie działań na listach</i>	85
5.6.	Listy zagnieżdżone i kopie głębokie	86
5.7.	Krotki	88
5.7.1.	<i>Podstawy krotek</i>	88
5.7.2.	<i>Jednoelementowe krotki wymagają przecinka</i>	89
5.7.3.	<i>Pakowanie i rozpakowywanie krotek</i>	90
5.7.4.	<i>Konwertowanie pomiędzy listami i krotkami</i>	91
5.8.	Zbiory	92
5.8.1.	<i>Działania na zbiorach</i>	92
5.8.2.	<i>Frozenset</i>	93
	Podsumowanie	93
Rozdział 6. Łańcuchy znaków		95
6.1.	Łańcuchy znaków jako sekwencje znaków	95
6.2.	Podstawowe działania na łańcuchach znaków	96
6.3.	Znaki specjalne i znaki ucieczki	96
6.3.1.	<i>Podstawowe sekwencje specjalne</i>	97
6.3.2.	<i>Numeryczne sekwencje specjalne i znaki Unicode</i>	97
6.3.3.	<i>Drukowanie i rozwijanie łańcuchów znaków ze znakami specjalnymi</i>	98
6.4.	Metody łańcuchów znaków	99
6.4.1.	<i>Metody split i join</i>	99
6.4.2.	<i>Konwersja łańcuchów znaków na liczby</i>	100
6.4.3.	<i>Usuwanie dodatkowych białych znaków</i>	101
6.4.4.	<i>Przeszukiwanie łańcuchów znaków</i>	102
6.4.5.	<i>Modyfikowanie łańcuchów znaków</i>	104
6.4.6.	<i>Zmienianie łańcuchów znaków przy użyciu operacji na listach</i>	105
6.4.7.	<i>Przydatne metody i stałe</i>	106
6.5.	Konwersja obiektów na łańcuchy znaków	107
6.6.	Korzystanie z metody format	108
6.6.1.	<i>Metoda format i parametry pozycyjne</i>	109
6.6.2.	<i>Metoda format i parametry wskazywane po nazwie</i>	109
6.6.3.	<i>Specyfikatory formatowania</i>	110
6.7.	Formatowanie łańcuchów znaków przy użyciu %	110
6.7.1.	<i>Korzystanie z sekwencji formatowania</i>	111
6.7.2.	<i>Parametry przekazywane przez nazwę i sekwencje formatujące</i>	112
6.8.	Interpolacja łańcuchów znaków	112
6.9.	Typ bytes	113
	Podsumowanie	115

Rozdział 7. Słowniki	117
7.1. Czym jest słownik?	117
7.2. Inne działania na słownikach	119
7.3. Liczenie słów	122
7.4. Co może być kluczem słownika?	123
7.5. Macierze rzadkie	124
7.6. Słowniki jako pamięć podręczna	125
7.7. Wydajność słowników	126
Podsumowanie	127
Rozdział 8. Przepływ sterowania	129
8.1. Pętla while	129
8.2. Instrukcja if-elif-else	130
8.3. Pętla for	132
8.3.1. Funkcja range	132
8.3.2. Ograniczenie funkcji range poprzez wartość początkową i krok	133
8.3.3. Używanie instrukcji break oraz continue w pętlach for	133
8.3.4. Pętla for i rozpakowywanie krotek	133
8.3.5. Funkcja enumerate	134
8.3.6. Funkcja zip	134
8.4. Listy i słowniki składane	135
8.4.1. Wyrażenia generatora	136
8.5. Instrukcje, bloki i wcięcia	136
8.6. Wartości i wyrażenia logiczne	139
8.6.1. Obiekty jako wartości logiczne	139
8.6.2. Porównania i operatory logiczne	140
8.7. Prosty program analizujący plik tekstowy	141
Podsumowanie	142
Rozdział 9. Funkcje	143
9.1. Podstawy definiowania funkcji	143
9.2. Opcje parametrów funkcji	144
9.2.1. Parametry pozycyjne	145
9.2.2. Przekazywanie argumentów przez nazwę parametru	146
9.2.3. Zmienna liczba argumentów	147
9.2.4. Łączenie technik przekazywania argumentów	148
9.3. Obiekty mutowalne jako argumenty	148
9.4. Zmienne lokalne, nielocalne i globalne	149
9.5. Przypisywanie funkcji do zmiennych	151
9.6. Wyrażenia lambda	152
9.7. Funkcje generatorów	152
9.8. Dekoratory	154
Podsumowanie	155

Rozdział 10. Moduły i zakresy	157
10.1. Czym jest moduł?	157
10.2. Pierwszy moduł	158
10.3. Instrukcja import	161
10.4. Ścieżka szukania modułów	161
10.4.1. <i>Gdzie umieszczać własne moduły</i>	162
10.5. Nazwy prywatne w modułach	163
10.6. Biblioteka i moduły zewnętrzne	164
10.7. Zasięg zmiennych i przestrzenie nazw w Pythonie	165
Podsumowanie	171
Rozdział 11. Programy w Pythonie	173
11.1. Tworzenie bardzo prostego programu	174
11.1.1. <i>Uruchamianie skryptu z wiersza poleceń</i>	174
11.1.2. <i>Argumenty wiersza poleceń</i>	175
11.1.3. <i>Przekierowywanie wejścia i wyjścia skryptu</i>	175
11.1.4. <i>Moduł argparse</i>	176
11.1.5. <i>Używanie modułu fileinput</i>	177
11.2. Tworzenie skryptów bezpośrednio wykonywalnych w systemie UNIX	179
11.3. Skrypty w systemach macOS	180
11.4. Możliwości wykonywania skryptów w systemach Windows	180
11.4.1. <i>Uruchamianie skryptu z wiersza poleceń lub poprzez PowerShell</i>	180
11.4.2. <i>Inne możliwości w systemach Windows</i>	181
11.5. Programy i moduły	181
11.6. Dystrybucja aplikacji w Pythonie	186
11.6.1. <i>Pakiety wheel</i>	186
11.6.2. <i>zipapp oraz pex</i>	186
11.6.3. <i>py2exe oraz py2app</i>	187
11.6.4. <i>Tworzenie programów wykonywalnych za pomocą freeze</i>	187
Podsumowanie	188
Rozdział 12. Praca z systemem plików	189
12.1. os i os.path a pathlib	190
12.2. Ścieżki i nazwy ścieżek	190
12.2.1. <i>Ścieżki bezwzględne i względne</i>	191
12.2.2. <i>Bieżący katalog roboczy</i>	192
12.2.3. <i>Poruszanie się po katalogach przy pomocy pathlib</i>	193
12.2.4. <i>Operacje na nazwach ścieżek</i>	193
12.2.5. <i>Operacje na nazwach ścieżek przy użyciu pathlib</i>	195
12.2.6. <i>Użyteczne stałe i funkcje</i>	196
12.3. Uzyskiwanie informacji o plikach	198
12.3.1. <i>Uzyskiwanie informacji o plikach przy użyciu scandir</i>	199
12.4. Więcej operacji w systemie plików	199
12.4.1. <i>Więcej operacji w systemie plików przy użyciu pathlib</i>	201
12.5. Obsługa wszystkich plików w części drzewa katalogów	202
Podsumowanie	203

Rozdział 13. Pisanie i czytanie plików	205
13.1. Otwieranie plików i obiektów typu file	205
13.2. Zamykanie plików	206
13.3. Otwieranie plików w różnych trybach	207
13.4. Funkcje do czytania i pisania danych tekstowych lub binarnych	207
13.4.1. <i>Używanie trybu binarnego</i>	209
13.5. Czytanie i pisanie przy pomocy pathlib	210
13.6. Operacje wejścia/wyjścia i przekierowania	210
13.7. Przekierowanie binarnych struktur danych i moduł struct	213
13.8. Serializacja obiektów do plików	215
13.8.1. <i>Argumenty przeciw serializacji</i>	217
13.9. Magazynowanie obiektów przy użyciu modułu shelve	218
Podsumowanie	220
Rozdział 14. Wyjątki	221
14.1. Wstęp do wyjątków	221
14.1.1. <i>Ogólna koncepcja błędów i obsługi wyjątków</i>	222
14.1.2. <i>Bardziej formalna definicja wyjątku</i>	224
14.1.3. <i>Obsługa różnych typów wyjątków</i>	225
14.2. Wyjątki w Pythonie	225
14.2.1. <i>Typy wyjątków w Pythonie</i>	226
14.2.2. <i>Zgłaszanie wyjątków</i>	228
14.2.3. <i>Łapanie i obsługa wyjątków</i>	229
14.2.4. <i>Definiowanie nowych wyjątków</i>	230
14.2.5. <i>Debugowanie programów przy użyciu instrukcji assert</i>	231
14.2.6. <i>Hierarchia dziedziczenia wyjątków</i>	232
14.2.7. <i>Przykład: program do pisania danych na dysku w Pythonie</i>	232
14.2.8. <i>Przykład: wyjątki w zwykłych przeliczeniach</i>	233
14.2.9. <i>Kiedy używać wyjątków?</i>	234
14.3. Managery kontekstu i słowo kluczowe with	235
Podsumowanie	236
CZĘŚĆ III. ZAAWANSOWANE CECHY JEZYKA	237
Rozdział 15. Klasy i programowanie zorientowane obiektowo	239
15.1. Definiowanie klas	239
15.1.1. <i>Wykorzystanie instancji klasy jako struktury lub rekordu</i>	240
15.2. Zmienne instancji	241
15.3. Metody	241
15.4. Zmienne klasy	243
15.4.1. <i>Zagwozdzka związana ze zmiennymi klasy</i>	244
15.5. Metody statyczne i metody klas	245
15.5.1. <i>Metody statyczne</i>	246
15.5.2. <i>Metody klas</i>	247
15.6. Dziedziczenie	248
15.7. Dziedziczenie i zmienne klasowe oraz zmienne instancji	250

15.8. Powtórka: podstawy klas w Pythonie	251
15.9. Zmienne i metody prywatne	253
15.10. @property i bardziej elastyczne zmienne instancji	254
15.11. Zasięg i przestrzenie nazw dla instancji klas	255
15.12. Destruktory i zarządzanie pamięcią	259
15.13. Wielodziedziczenie	260
Podsumowanie	262
Rozdział 16. Wyrażenia regularne	263
16.1. Co to jest wyrażenie regularne?	263
16.2. Wyrażenia regularne ze znakami specjalnymi	264
16.3. Wyrażenia regularne i łańcuchy znaków	265
16.3.1. Raw stringi	266
16.4. Uzyskiwanie dostępu do dopasowanego tekstu w łańcuchu znaków	267
16.5. Zastępowanie tekstu wyrażeniem regularnym	270
Podsumowanie	272
Rozdział 17. Typy danych jako obiekty	273
17.1. Typy również są obiektami	273
17.2. Korzystanie z typów	274
17.3. Typy i klasy zdefiniowane przez użytkownika	274
17.4. Duck typing	276
17.5. Czym jest specjalny atrybut metody?	277
17.6. Obiekty zachowujące się jak listy	278
17.7. Atrybut metody __getitem__	279
17.7.1. Jak to działa?	280
17.7.2. Implementacja kompletu funkcjonalności listy	281
17.8. Obiekt o wszystkich możliwościach listy	281
17.9. Klasy pochodne od typów wbudowanych	283
17.9.1. Pochodne od listy	283
17.9.2. Pochodne klasy UserList	284
17.10. Kiedy korzystać ze specjalnych atrybutów metod?	285
Podsumowanie	286
Rozdział 18. Pakiety	287
18.1. Czym jest pakiet?	287
18.2. Pierwszy przykład	288
18.3. Konkretny przykład	289
18.3.1. Pliki __init__ w pakietach	291
18.3.2. Podstawowe użycie pakietu matplotlib	291
18.3.3. Ładowanie subpakietów i submodułów	291
18.3.4. Instrukcja import wewnątrz pakietów	292
18.4. Atrybut __all__	293
18.5. Właściwe korzystanie z pakietów	294
Podsumowanie	295

Rozdział 19. Korzystanie z bibliotek Pythona	297
19.1. „Wszystko w standardzie” — biblioteka standardowa	298
19.1.1. Praca z różnymi typami danych	298
19.1.2. Operacje na plikach i pamięci	298
19.1.3. Dostęp do usług systemu operacyjnego	300
19.1.4. Korzystanie z protokołów i formatów internetu	300
19.1.5. Narzędzia do tworzenia i debugowania oraz usługi uruchomieniowe	301
19.2. Wyjście poza bibliotekę standardową	301
19.3. Dodawanie kolejnych bibliotek w Pythonie	302
19.4. Instalowanie bibliotek Pythona przy użyciu pip oraz venv	302
19.4.1. Instalacja z flagą --user	303
19.4.2. Środowiska wirtualne	303
19.5. PyPI (czyli The Cheese Shop)	304
Podsumowanie	304
CZĘŚĆ IV. PRACA Z DANymi	305
Rozdział 20. Podstawy obsługi plików	307
20.1. Problem: niekończący się napływ plików z danymi	307
20.2. Scenariusz: dane produktowe z piekła	308
20.3. Więcej organizacji	310
20.4. Oszczędzanie miejsca: kompresja i sprzątanie	311
20.4.1. Kompresja	311
20.4.2. Sprzątanie plików	312
Podsumowanie	314
Rozdział 21. Procesowanie plików danych	315
21.1. Witamy w ETL	315
21.2. Czytanie plików tekstowych	316
21.2.1. Kodowanie tekstu: ASCII, Unicode itp.	316
21.2.2. Tekst nieustrukturyzowany	318
21.2.3. Pliki płaskie podzielone znakami specjalnymi	320
21.2.4. Moduł csv	322
21.2.5. Czytanie pliku CSV jako listy słowników	324
21.3. Pliki Excel	324
21.4. Czyszczenie danych	326
21.4.1. Czyszczenie	326
21.4.2. Sortowanie	327
21.4.3. Problemy i pułapki czyszczenia danych	328
21.5. Pisanie plików z danymi	329
21.5.1. CSV i pliki dzielone znakami specjalnymi	329
21.5.2. Zapisywanie plików Excel	330
21.5.3. Pakowanie plików danych	331
Podsumowanie	331

Rozdział 22. Dane w sieci	333
22.1. Pobieranie plików	333
22.1.1. Korzystanie z Pythona do pobierania plików z serwera FTP	334
22.1.2. Pobieranie plików przy użyciu SFTP	335
22.1.3. Pobieranie plików przy użyciu HTTP/HTTPS	336
22.2. Pobieranie danych przez API	337
22.3. Ustrukturyzowane formaty danych	339
22.3.1. Dane w formacie JSON	339
22.3.2. Dane XML	342
22.4. Szczytywanie danych z sieci WWW	347
Podsumowanie	351
Rozdział 23. Przechowywanie plików	353
23.1. Relacyjne bazy danych	354
23.1.1. Bazodanowe API Pythona	354
23.2. SQLite: korzystanie z bazy danych SQLite	354
23.3. Używanie MySQL, PostgreSQL i innych relacyjnych baz danych	357
23.4. Ułatwienie pracy z bazą danych — ORM	357
23.4.1. SQLAlchemy	358
23.4.2. Wykorzystanie Alembic do zmian struktury bazy danych	361
23.5. Nierelacyjne bazy danych	364
23.6. Klucz-wartość i Redis	364
23.7. Dokumenty w MongoDB	367
Podsumowanie	370
Rozdział 24. Badanie danych	371
24.1. Narzędzie do badania danych	371
24.1.1. Zalety Pythona w zakresie obsługi danych	371
24.1.2. Python może być lepszy niż arkusz kalkulacyjny	372
24.2. Notatnik Jupyter	372
24.2.1. Uruchomienie jądra	373
24.2.2. Wykonanie kodu w komórce	373
24.3. Python i pandas	375
24.3.1. Dlaczego mógłbyś chcieć używać pandas?	375
24.3.2. Instalacja pandas	375
24.3.3. Ramki danych	376
24.4. Czyszczenie danych	377
24.4.1. Ładowanie i zachowywanie danych w pandas	377
24.4.2. Czyszczenie danych i ramki danych	379
24.5. Agregowanie danych i manipulowanie nimi	381
24.5.1. Łączenie ramek danych	382
24.5.2. Wybieranie danych	383
24.5.3. Grupowanie i agregacja	384
24.6. Obrazowanie danych	385
24.7. Kiedy nie używać biblioteki pandas?	386
Podsumowanie	387

<i>Studium przypadku</i>	389
Pobranie danych	389
Parsowanie danych dat pomiarów	392
Wybór stacji na podstawie długości i szerokości geograficznej	393
Wybór stacji i uzyskanie jej metadanych	395
Pozyskanie i sparsowanie danych pogodowych	397
<i>Pozyskanie danych</i>	397
<i>Parsowanie danych pogodowych</i>	397
Zapisywanie danych pogodowych do bazy danych (opcjonalne)	400
Wybieranie i obrazowanie danych	401
Użycie pandas do tworzenia wykresu	401
<i>Dodatek A. Przewodnik po dokumentacji Pythona</i>	403
<i>Dodatek B. Odpowiedzi do ćwiczeń</i>	425
<i>Skorowidz</i>	467

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 określeniem 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.

SYBKI 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 *między* 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 nieznannej 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 indeksem 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]` robi 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 unikać 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 znajdują 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.

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

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

SYBKI 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.

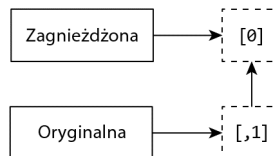
Najczęściej na tym można poprzestać, jeśli chodzi o posługiwanie się listami zagnieżdżonymi. Czasami jednak można napotkać pewne szczególne problemy z tymi strukturami danych, zwłaszcza jeśli chodzi o odwoływanie się poprzez zmienną do obiektu oraz sposób, w jaki pewne obiekty (np. listy) mogą być modyfikowane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

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

Rysunek 5.1 obrazuje ten przykład.

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]
```



Rysunek 5.1. Lista z pierwszym elementem referującym do zagnieżdżonej listy

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.

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.

Listy, na które wskazują zmienne `oryginalna` oraz `płytką`, są połączone. Zmiana wartości na zagnieżdżonej liście z poziomu którejkolwiek z nich skutkuje zmianami dla obu.

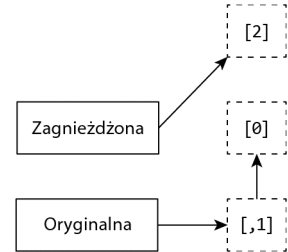
```
>>> płytka[1] = 2
>>> płytka
[[0], 2]
>>> oryginalna
[[0], 1]
>>> płytka[0][0] = 'zero'
>>> oryginalna
[['zero'], 1]
```

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

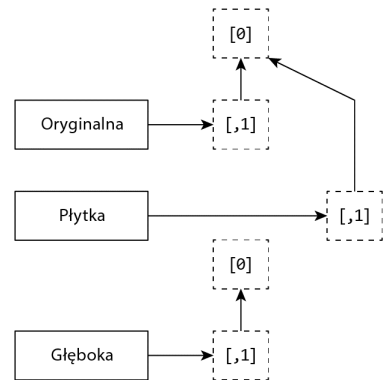
```
>>> głęboka[0][0] = 5
>>> głęboka
[[5], 1]
>>> oryginalna
[['zero'], 1]
```

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.



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



Rysunek 5.3. Płytką kopia nie kopiuje zagnieżdżonych list

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 niezmiennie 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_wejsciowy:
...     for wiersz in plik_wejsciowy:
...         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

A

agregacja danych, 384
agregowanie danych, 381
Alembic, 361, 364
API, 337
 pobieranie danych, 337
argumenty
 nazwane, 147
 obiekty mutowalne, 148
 pozycyjne, 147
 wiersza poleceń, 175
ASCII, 316
atrybut
 __all__, 293
 __getitem__, 279
atrybuty specjalne metod,
 277, 285

B

badanie danych, 371–387
baza danych
 MongoDB, 367
 ORM, 357
 Redis, 364
 SQLite, 354
bazodanowe API Pythona, 354
bazy danych
 nierelacyjne, 364
 relacyjne, 357
 sieciowe, 364
 zmiana struktury, 361
białe znaki, 63, 101
biblioteka, 164, 297
pandas, 375, 386
 ładowanie danych, 377
 zachowywanie danych,
 377
pathlib, 190, 193, 201

bieżący katalog, 193
czytanie i pisanie, 210
funkcje i atrybuty, 204
operacje na nazwach
 ścieżek, 195
operacje w systemie
 plików, 201
requests, 341
standardowa, 298
bloki, 63, 136
błędy, 222

C

CSV, comma-separated
values, 321
czyszczenie danych, 326,
328, 377, 379

D

dane, 371–387
 od użytkownika, 72
 studium przypadku,
 389–402
 w sieci, 333
 XML, 342
debugowanie, 231
definiowanie
 funkcji, 55, 143
 klas, 239
 wyjątków, 224, 230
dekoratory, 154
deskryptywność, 415
destrukторы, 259
długość linii, 410
dodawanie bibliotek, 302
dokumentacja, 403, 414
dostęp
 do dokumentacji, 403
 do dopasowanego tekstu,
 267

do usług, 300
do zmiennych instancji,
 254
drzewo katalogów, 202
duck typing, 276
dystrybucja aplikacji, 186
działania
 na listach, 83
 na łańcuchach znaków,
 96
 na słownikach, 119
 na zbiorach, 92
 w systemie plików, 203,
 204
dziedziczenie, 248, 250

E

ETL, extract-transform-load,
 315, 331
etykiety, 66

F

format
 CSV, 321
 JSON, 339
formatowanie, 110
 łańcuchów znaków, 110
formaty internetowe, 300
frozenset, 93
FTP, File Transfer Protocol,
 334
funkcja, 55
 enumerate, 134
 max, 84
 min, 84
 print, 113
 range, 132
 krok, 133
 wartość początkowa, 133

funkcja
 sorted, 83
 zip, 134

funkcje
 definiowanie, 143
 do czytania i pisania, 207
 generatorów, 152
 liczb zespolonych, 71
 liczbowe, 70
 wbudowane, 70
 zaawansowane, 70
 opcje parametrów, 144
 ścieżek, 196
 zmienna liczba
 argumentów, 147

G

generator, 136
 generowanie stron HTML,
 404
 getter, 254
 grupowanie danych, 384

H

hierarchia dziedziczenia
 wyjątków, 232

I

IDLE, 39, 40, 41
 importy, 410
 indeksy list, 76
 informacje o plikach, 198,
 199
 inicjalizacja listy, 83
 instalacja, 37
 bibliotek, 302
 pandas, 375
 instrukcja, 136
 assert, 231
 break, 133
 class, 239
 continue, 133
 if-elif-else, 53, 130
 import, 161, 292
 try, 229
 interpolacja łańcuchów
 znaków, 112

J

Jupyter, 372

K

catalog roboczy, 192
 klasa UserList, 284
 klasy
 definiowanie, 239
 destrukторы, 259
 dziedziczenie, 248
 inicjalizatory, 259
 mapowanie obiektów
 tabel, 360
 metody, 241
 pochodne, 283
 wielodziedziczenie, 260
 zmienne, 243, 262

klauzula else, 230
 klucz-wartość, 364
 kodowanie tekstu, 316
 komentarze, 65, 414
 blokowe, 414
 liniowe, 414

konsola, 42
 kontenery, 66
 kontrola przepływu
 sterowania, 53
 konwencje
 kodowania, 73
 nazewnicze, 415, 416

konwersja
 łańcuchów znaków, 100
 obiektów, 107

kopie głębokie, 86
 krotki, 49, 88
 jednoelementowe, 89
 konwertowanie, 91
 pakowanie, 90
 rozpakowywanie, 133

L

liczby, 46, 68
 zespolone, 70
 liczenie słów, 122
 lista słowników, 324

listy, 48, 76, 135
 działania, 83, 85
 implementacja
 funkcjonalności, 281
 indeksy, 76
 inicjalizacja, 83
 minimalna wartość
 elementu, 84
 modyfikowanie, 78
 przeszukiwanie, 84
 sortowanie, 80
 wystąpienia elementu, 85
 zagnieżdżone, 86

Ł

łańcuchy znaków, 50, 68, 95
 dostęp do dopasowanego
 tekstu, 267
 drukowanie, 98
 działania, 96
 formatowanie, 110
 interpolacja, 112
 konwersja na liczby, 100
 konwersja obiektów, 107
 metody, 99
 modyfikowanie, 104
 operacje, 107
 przeszukiwanie, 102
 rozwijanie, 98
 usuwanie białych
 znaków, 101
 ze znakami specjalnymi,
 98
 zmienianie, 105
 łączenie ramek danych, 382

M

macierze rzadkie, 124
 magazynowanie obiektów,
 218
 managery kontekstu, 235
 mapowanie
 obiektowo-relacyjne,
 ORM, 361, 370
 obiektów tabel na klasy,
 360

- mechanizm
 - sortowania, 81
 - wyjątków, 224
 - metadane, 395
 - metoda, 241
 - `__getitem__`, 279
 - `__init__`, 262
 - `count`, 85
 - `format`, 108, 109
 - `index`, 84
 - `join`, 99
 - `split`, 99
 - metody
 - specjalne atrybuty, 277, 285
 - klas, 246, 247
 - łańcuchów znaków, 99, 107
 - prywatne, 253
 - statyczne, 245, 262
 - wirtualne, 262
 - moduł, 57, 157, 181
 - `argparse`, 176
 - `csv`, 322
 - `fileinput`, 177
 - `math`, 71
 - `os`, 190
 - `os.path`, 190
 - `pickle`, 220
 - `pydoc`, 404
 - `shelve`, 218, 220
 - `struct`, 213, 220
 - moduły
 - liczbowe i matematyczne, 299
 - nazwy prywatne, 163
 - obsługujące formaty internetowe, 300
 - obsługujące protokoły, 300
 - plików, 299
 - ścieżka szukania, 161
 - typów danych, 299
 - umieszczanie, 162
 - usług, 298
 - zewnętrzne, 164
 - związane z kodowaniem, 301
 - związane z systemem operacyjnym, 300
 - modyfikacje
 - bazy danych, 364
 - list, 78
 - łańcuchów znaków, 104
 - MongoDB, 367
 - skalowalność, 370
 - uruchomienie serwera, 368
 - MySQL, 357
- ## N
- narzędzie
 - Alembic, 370
 - do badania danych, 371
 - `freeze`, 187
 - `pip`, 302
 - nazewnictwo, 415
 - nazwy ścieżek, 190, 193
 - nierelacyjne bazy danych, 364
 - notatnik Jupyter, 372
- ## O
- obiekty
 - jako listy, 278, 281
 - jako wartości logiczne, 139
 - mutowalne, 148
 - plików, 52, 205
 - typy danych, 273
 - widoków, 120
 - obrazowanie danych, 385, 401
 - obsługa
 - argumentów, 147
 - błędów, 56
 - danych, 371
 - plików, 202
 - ścieżek plików, 190
 - wyjątków, 222, 225, 229
 - odpowiedzi do ćwiczeń, 425–466
 - okno konsoli, 41
 - OOP, object-oriented programming, 58
 - opcje parametrów, 144
 - operacje
 - na bazie `sqlite3`, 356
 - wejścia/wyjścia, 210
 - operator, 73
 - `*`, 83
 - `in`, 83
 - operatory logiczne, 140
 - ORM, model obiektowo-relacyjny, 357, 370
 - otwieranie plików, 205
- ## P
- pakiet, 287, 290
 - `matproj`, 288, 291
 - `pex`, 186
 - `py2app`, 187
 - `py2exe`, 187
 - `wheel`, 186
 - `zipapp`, 186
 - pakiety
 - instrukcja `import`, 292
 - korzystanie, 294
 - ładowanie, 291
 - pliki `__init__`, 291
 - pamięć
 - operacje, 298
 - podręczna, 125
 - parametr `self`, 262
 - parametry
 - pozycyjne, 109, 145
 - przekazywane przez nazwę, 109, 112
 - parsowanie danych, 392, 397
 - pętla
 - `for`, 54, 132, 133
 - `while`, 54, 129
 - pierwszy program, 42
 - pisanie danych na dysku, 232
 - plik
 - `jedyna1.txt`, 178
 - `kolo.py`, 246
 - `l_na_n.py`, 183
 - `matematyka.py`, 158
 - `mio.py`, 212
 - `modtest.py`, 164
 - `modul_kolor.py`, 277
 - `moje_moduly.pth`, 163

- plik
 - mozolne_przeliczenia.py, 216
 - nonlocal.py, 150
 - opcje.py, 176
 - pliki_01.py, 309–313
 - prognoza.html, 349
 - skrypt1.py, 174–181
 - test.html, 347
 - test_zakresu.py, 168
 - zamień.py, 175
 - zk.py, 257
 - pliki, 198, 307
 - __init__, 291
 - archiwizacja, 311
 - CSV, 329, 331
 - danych, 315
 - Excel, 324, 330
 - JSON, 339
 - kompresja, 311
 - napływ danych, 307
 - odczyt, 316
 - odczyt formatu CSV, 324
 - operacje, 298
 - pakowanie, 331
 - plaskie, 320
 - pobieranie, 333
 - pomocy Windows, 405
 - procesowanie, 315
 - przechowywanie, 310
 - serializacja obiektów, 215
 - sprzątanie, 311
 - tekstowe, 316
 - tryb binarny, 209
 - tryby otwierania, 205, 207
 - XLS/XLSX, 331
 - z danymi, 329
 - zamykanie, 206
 - znaki specjalne, 320
 - pobieranie
 - danych, 389
 - przez API, 337
 - z sieci WWW, 347
 - dokumentacji, 406
 - plików, 333
 - FTP, 334
 - HTTP/HTTPS, 336
 - SFTP, 335
 - polecenie
 - help, 404
 - os.scandir, 199
 - pomoc, 404
 - porównania, 140
 - PostgreSQL, 357
 - PowerShell, 180
 - powłoka interaktywna, 40
 - preskryptywność, 416
 - procedura, 144
 - program analizujący plik tekstowy, 141
 - programowanie
 - zorientowane obiektowo, OOP, 58, 239
 - programy, 173, 181
 - przekierowywanie wejścia, 175
 - tworzenie, 174
 - protokoły, 300
 - przechowywanie wersji, 415
 - przeglądanie dokumentacji, 404
 - przekazywanie argumentów, 148
 - przez nazwę parametru, 146
 - przekierowanie, 210, 213
 - wejścia, 175
 - przeptyw sterowania, 129
 - przeźrzenie nazw, 165, 255
 - przeszukiwanie listy, 84
 - przypisania, 65
 - funkcji do zmiennych, 151
 - puste linie, 410
 - PyPI, Python Package Index, 304
 - Python
 - wady, 33
 - zalety, 30
- ## R
- ramki danych, 376, 379
 - raw stringi, 266
 - Redis, 364
 - uruchomienie serwera, 365
 - wygasanie wartości, 366
 - rekordy, 240
 - relacyjne bazy danych, 353, 357
 - repozytorium kodu, 304
- ## S
- sekwencje
 - formatowania, 111, 112
 - specjalne, 97
 - znaków, 95
 - serializacja obiektów, 215
 - serwer
 - bazy Redis, 365
 - dokumentacji, 405
 - FTP, 334
 - setter, 254
 - SFTP, SSH File Transfer Protocol, 335
 - sieciowe bazy danych, 364
 - skrypty w systemie
 - macOS, 180
 - UNIX, 179
 - Windows, 180
 - słowniki, 51, 117
 - jako pamięć podręczna, 125
 - klucze, 123
 - operacje, 119, 122
 - składane, 135
 - wydajność, 126
 - słowo kluczowe
 - with, 56, 235
 - yield, 153
 - sortowanie, 327
 - list, 80
 - spacje, 409
 - specyfikatory formatowania, 110
 - SQLAlchemy, 358
 - SQLite, 354
 - stacje badawcze, 393
 - stałe, 196
 - status funkcji, 222
 - struktura
 - kodu, 409
 - wcięć, 63
 - struktury, 240
 - studium przypadku, 389–402

styl kodu, 407, 408
 style nazewnictwa, 415
 system plików, 189
 funkcje, 203, 204
 operacje, 199

Ś

ścieżka szukania modułów,
 161
 ścieżki, 190
 bezwzględne, 191
 funkcje, 196
 operacje na nazwach,
 193, 195
 stałe, 196
 wykorzystanie pathlib,
 195
 względne, 191
 środowisko wirtualne, 303

T

tablice, 76
 tabulacja, 409
 tekst nieustrukturyzowany,
 318
 tryb
 binarny, 209
 konsolowy, 39, 41
 tworzenie
 modułów, 57
 programu, 174
 wykresu, 401
 typ bytes, 113
 typy
 danych jako obiekty, 273
 wbudowane, 46
 wyjątków, 226
 zdefiniowane przez
 użytkownika, 274

U

Unicode, 316
 UNIX
 tworzenie skryptów, 179
 uruchamianie
 jądra, 373
 skryptu, 174, 180

ustrukturyzowane formaty
 danych, 339
 UTF-8, 316
 uzyskiwanie
 danych, 72
 pomocy, 404
 używanie konsoli, 42

W

wartości, 139
 domyślne, 145
 logiczne, 53
 wartość
 NaN, 379
 None, 72
 wbudowane
 funkcje liczbowe, 70
 operatory, 73
 wcięcia, 63, 136, 409
 wielodziedziczenie, 260, 262
 wiersz poleceń
 argumenty, 175
 uruchamianie skryptu,
 174, 180
 właściwości, 262
 wybieranie danych, 383
 wybór stacji badawczej, 393,
 395
 wyjątki, 55, 221, 225
 definiowanie, 230
 formalna definicja, 224
 hierarchia dziedziczenia,
 232
 obsługa, 225, 229
 typy, 226
 używanie, 234
 w przeliczeniach, 233
 zgłaszanie, 228
 wyjścia skryptu, 175
 wykonanie kodu w komórce,
 373
 wyrażenia, 53, 67
 generatora, 136
 lambda, 152
 logiczne, 139
 regularne, 263
 dostęp do tekstu, 267
 flagi, 272

łańcuchy znaków, 265
 zastępowanie tekstu,
 270
 ze znakami
 specjalnymi, 264

X

XML, eXtensible Markup
 Language, 342

Z

zachowywanie danych, 353
 zapisywanie danych, 400
 zarządzanie pamięcią, 259
 zasięg, 255
 zmiennych, 165
 zbiory, 52, 92
 operacje, 92
 Zen Pythona, 422
 zgłaszanie wyjątków, 228
 zintegrowane środowisko
 programistyczne IDLE, 40
 zmiana struktury bazy
 danych, 361
 zmienna liczba argumentów,
 147
 zmienne, 65
 globalne, 149
 instancji, 241, 250, 254,
 262
 lokalne, 149
 nielokalne, 149
 prywatne, 253
 zasięg, 165
 znak
 %, 110
 zachęty, 42
 znaki
 białe, 411
 specjalne, 96, 264, 320
 ucieczki, 96
 Unicode, 97

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

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 obszernie 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	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI Sięgnij po więcej! ▶ 
 helion.pl	 SZKOLENIA AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	ISBN 978-83-283-3771-8  9 788328 337718
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 79,00 zł