

O'REILLY®

Wydanie VI

Python

Wprowadzenie



Helion 

Mark Lutz

Tytuł oryginału: Learning Python: Powerful Object-Oriented Programming, 6th Edition

Tłumaczenie: Anna Mizerska z wykorzystaniem fragmentów poprzedniego wydania w przekładzie
Grzegorza Kowalczyka, Andrzeja Watraka, Anny Trojan i Marka Pętlickiego

ISBN: 978-83-289-2942-5

© 2025 Helion S.A.

Authorized Polish translation of the English edition of *Learning Python, 6th Edition*
ISBN 9781098171308 © 2025 Mark Lutz

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności
za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

helion.pl/user/opinie/pytho6

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: helion.pl (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa	29
<hr/>	
Część I. Wprowadzenie	35
1. Pytania i odpowiedzi dotyczące Pythona	37
Dlaczego ludzie używają Pythona?	37
Jakość oprogramowania	39
Wydajność programistów	39
Czy Python jest językiem skryptowym?	40
Jakie są zatem wady języka Python?	41
Kto dzisiaj używa Pythona?	43
Co mogę zrobić za pomocą Pythona?	43
Programowanie systemowe	44
Graficzne interfejsy użytkownika (GUI) i interfejsy użytkownika (UI)	44
Skrypty internetowe	44
Integracja komponentów	45
Dostęp do baz danych	45
Szybkie prototypowanie	46
Programowanie numeryczne i naukowe	46
I więcej: sztuczna inteligencja, gry, przetwarzanie obrazu, wyszukiwanie danych, testowanie, Excel, aplikacje...	47
Jakie są techniczne mocne strony Pythona?	47
Jest zorientowany obiektowo i funkcyjny	47
Jest darmowy i otwarty	48
Jest przenośny	48
Ma duże możliwości	49
Można go łączyć z innymi językami	50
Jest względnie łatwy w użyciu	51
Jest względnie łatwy do nauczenia się	51
Podsumowanie rozdziału	52
Sprawdź swoją wiedzę — quiz	52
Sprawdź swoją wiedzę — odpowiedzi	53

2. Jak Python wykonuje programy?	54
Wprowadzenie do interpretera Pythona	54
Wykonywanie programu	55
Z punktu widzenia programisty	55
Z punktu widzenia Pythona	56
Warianty modeli wykonywania	60
Alternatywne implementacje Pythona	60
Samodzielne pliki wykonywalne	64
Przyszłe możliwości	65
Podsumowanie rozdziału	65
Sprawdź swoją wiedzę — quiz	65
Sprawdź swoją wiedzę — odpowiedzi	66
3. Jak uruchamia się programy?	67
Instalacja Pythona	67
Interaktywny kod	68
Uruchamianie interaktywnego środowiska REPL	68
Gdzie uruchamiać programy — katalogi z kodem źródłowym	70
Czego nie wpisywać — znaki zachęty i komentarze	71
Inne środowiska REPL Pythona	72
Interaktywne wykonywanie kodu	73
Do czego służy sesja interaktywna?	74
Pliki źródłowe	76
Pierwszy skrypt	77
Wykonywanie plików z poziomu wiersza poleceń powłoki	78
Sposoby użycia wiersza poleceń	79
Inne sposoby uruchamiania plików	81
Klikanie ikon plików	81
Interfejs użytkownika środowiska IDLE	82
Inne środowiska IDE	84
Aplikacje na smartfony	84
WebAssembly dla przeglądarek	84
Notatniki Jupyter do celów naukowych	85
Kompilatory AOT dla zwiększenia szybkości	85
Uruchamianie kodu w kodzie	86
Inne opcje wykonywania kodu	91
Jaką opcję wybrać?	92
Podsumowanie rozdziału	92
Sprawdź swoją wiedzę — quiz	93
Sprawdź swoją wiedzę — odpowiedzi	93
Sprawdź swoją wiedzę — ćwiczenia do części I	94

4. Wprowadzenie do obiektów Pythona	101
Hierarchia pojęć w Pythonie	101
Dlaczego korzystamy z obiektów wbudowanych?	102
Najważniejsze typy danych w Pythonie	103
Liczby	105
Łańcuchy znaków	107
Operacje na sekwencjach	107
Niezmienność	109
Metody specyficzne dla typu	110
Uzyskiwanie pomocy	112
Inne sposoby kodowania łańcuchów znaków	113
Ciągi znaków w formacie Unicode	114
Listy	116
Operacje na typach sekwencyjnych	116
Operacje specyficzne dla typu	117
Sprawdzanie granic	117
Zagnieżdżanie	118
Listy składane	119
Słowniki	120
Operacje na odwzorowaniach	121
Zagnieżdżanie raz jeszcze	122
Brakujące klucze — testowanie za pomocą if	124
Sortowanie kluczy — pętla for	126
Krotki	127
Do czego służą krotki?	128
Pliki	128
Pliki tekstowe Unicode i binarne	130
Inne narzędzia podobne do plików	131
Inne typy podstawowe	131
Zbiory	131
Wartości logiczne i obiekt None	132
Typy	132
Podpowiedzi typów	133
Klasy definiowane przez użytkownika	134
I wszystko inne	135
Podsumowanie rozdziału	135
Sprawdź swoją wiedzę — quiz	136
Sprawdź swoją wiedzę — odpowiedzi	136

5. Typy liczbowe	138
Podstawy typów liczbowych Pythona	138
Literały liczbowe	139
Wbudowane narzędzia liczbowe	141
Operatory wyrażeń Pythona	141
Połączone operatory stosują się do priorytetów	143
Podwyrażenia grupowane są w nawiasach	144
Pomieszane typy poddawane są konwersji	144
Wprowadzenie: przeciążanie operatorów i polimorfizm	145
Liczby w akcji	146
Zmienne i podstawowe wyrażenia	146
Formaty wyświetlania liczb	148
Operatory porównania	149
Operatory dzielenia	151
Precyzja liczb całkowitych	153
Liczby zespolone	154
Notacja szesnastkowa, ósemkowa i dwójkowa	154
Operacje na poziomie bitów	156
Znaki podkreślenia jako separatory w liczbach	158
Inne wbudowane narzędzia numeryczne	160
Inne typy liczbowe	162
Typ Decimal (liczby dziesiętne)	162
Typ Fraction (liczby ułamkowe)	164
Zbiory	166
Wartości Boolean	173
Rozszerzenia numeryczne	174
Podsumowanie rozdziału	174
Sprawdź swoją wiedzę — quiz	175
Sprawdź swoją wiedzę — odpowiedzi	175
6. Wprowadzenie do typów dynamicznych	177
Sprawa brakujących deklaracji typu	177
Zmienne, obiekty i referencje	178
Typy powiązane są z obiektami, a nie ze zmiennymi	180
Obiekty są uwalniane	181
Referencje współdzielone	183
Referencje współdzielone a modyfikacje w miejscu	184
Referencje współdzielone a równość	186
Typy dynamiczne są wszędzie	188
Podpowiedzi typów: opcjonalne, nieużywane i po co?	189
Podsumowanie rozdziału	190
Sprawdź swoją wiedzę — quiz	191
Sprawdź swoją wiedzę — odpowiedzi	191

7. Łańcuchy znaków	193
Łańcuchy znaków — podstawy	194
Literały łańcuchów znaków	196
Łańcuchy znaków w apostrofach i cudzysłowach są tym samym	196
Sekwencje ucieczki reprezentują znaki specjalne	197
Surowe łańcuchy znaków blokują sekwencje ucieczki	201
Potrójne cudzysłowy i apostrofy kodują łańcuchy znaków będące wielowierszowymi blokami	202
Łańcuchy znaków w akcji	204
Podstawowe operacje	205
Indeksowanie i wycinki	206
Narzędzia do konwersji łańcuchów znaków	211
Modyfikowanie łańcuchów znaków — część I: działania na sekwencjach	213
Metody łańcuchów znaków	214
Składnia wywoływania metod	214
Metody typów znakowych	215
Modyfikowanie łańcuchów znaków — część II: metody łańcuchów znaków	216
Więcej metod łańcuchów znaków — analiza składniowa tekstu	219
Inne często używane metody łańcuchów znaków	220
Formatowanie łańcuchów znaków — triathlon	221
Opcje formatowania łańcuchów znaków	221
Formatowanie z użyciem wyrażeń formatujących	222
Formatowanie łańcuchów z użyciem metody format	227
Podstawy	227
Zaawansowana składnia wywołań metody format	230
Literał formatowania f-string	234
A zwięźczą jest...	239
Generalne kategorie typów	241
Typy z jednej kategorii współdzielą zbiory operacji	241
Typy mutowalne można modyfikować w miejscu	242
Podsumowanie rozdziału	242
Sprawdź swoją wiedzę — quiz	243
Sprawdź swoją wiedzę — odpowiedzi	243
8. Listy i słowniki	245
Listy	245
Listy w akcji	248
Podstawowe operacje na listach	248
Indeksowanie i wycinki	249
Modyfikacja list w miejscu	250
Iteracje po listach i składanie list	255
Słowniki	258

Słowniki w akcji	260
Podstawowe operacje na słownikach	260
Modyfikacja słowników w miejscu	262
Inne metody słowników	263
Inne sposoby tworzenia słowników	264
Słowniki składane	266
Kolejność wstawiania kluczy	268
Operator „sumy” dla słowników	269
Przykład: baza danych o książkach	270
Uwagi na temat korzystania ze słowników	272
Podsumowanie rozdziału	280
Sprawdź swoją wiedzę — quiz	281
Sprawdź swoją wiedzę — odpowiedzi	281
9. Krotki, pliki i wszystko inne	284
Krotki	285
Krotki w akcji	285
Dlaczego istnieją listy i krotki?	289
Repetitorium: rekordy — krotki nazwane	290
Pliki	291
Otwieranie plików	292
Wykorzystywanie plików	293
Pliki w akcji	295
Pliki tekstowe i binarne — krótka historia	297
Przechowywanie obiektów Pythona w plikach i ich przetwarzanie	299
Przechowywanie natywnych obiektów Pythona — moduł pickle	301
Przechowywanie obiektów Pythona w formacie JSON	302
Przechowywanie obiektów za pomocą innych narzędzi	303
Menedżery kontekstu plików	304
Inne narzędzia powiązane z plikami	304
Przegląd i podsumowanie podstawowych typów obiektów	306
Elastyczność obiektów	307
Referencje a kopie	308
Porównania, testy równości i prawda	310
Porównywanie słowników	313
Prawda czy fałsz, czyli znaczenie True i False w Pythonie	314
Hierarchie typów Pythona	316
Obiekty typów	316
Inne typy w Pythonie	318
Pułapki typów wbudowanych	319
Przypisanie tworzy referencje, nie kopie	319
Powtórzenie dodaje jeden poziom zagłębienia	319

Uwaga na cykliczne struktury danych	320
Typów niemutowalnych nie można modyfikować w miejscu	321
Podsumowanie rozdziału	321
Sprawdź swoją wiedzę — quiz	322
Sprawdź swoją wiedzę — odpowiedzi	322
Sprawdź swoją wiedzę — ćwiczenia do części II	323

Część III. Instrukcje i składnia **327**

10. Wprowadzenie do instrukcji Pythona **329**

Raz jeszcze o hierarchii pojęciowej języka Python	329
Instrukcje Pythona	330
Historia dwóch if	332
Co dodaje Python?	332
Co usuwa Python?	333
Skąd bierze się składnia z użyciem wcięć?	335
Kilka przypadków specjalnych	337
Szybki przykład: interaktywne pętle	340
Prosta pętla interaktywna	340
Wykonywanie obliczeń na danych wpisywanych przez użytkownika	342
Obsługa błędów poprzez sprawdzanie danych wejściowych	342
Obsługa błędów za pomocą instrukcji try	344
Kod zagnieżdżony na trzy poziomy głębokości	346
Podsumowanie rozdziału	346
Sprawdź swoją wiedzę — quiz	347
Sprawdź swoją wiedzę — odpowiedzi	347

11. Przypisania, wyrażenia i wyświetlanie **348**

Instrukcje przypisania	348
Formy instrukcji przypisania	349
Podstawowe przypisanie	351
Przypisanie sekwencji	351
Rozszerzona składnia rozpakowania sekwencji w Pythonie 3.x	354
Przypisanie z wieloma celami	360
Przypisania rozszerzone	361
Wyrażenia przypisania nazwanego	364
Reguły dotyczące nazw zmiennych	367
Instrukcje wyrażeń	371
Instrukcje wyrażeń i modyfikacje w miejscu	372
Polecenia print	373
Funkcja print	373
Przekierowanie strumienia wyjściowego	376

Podsumowanie rozdziału	380
Sprawdź swoją wiedzę — quiz	381
Sprawdź swoją wiedzę — odpowiedzi	381
12. Testy if i reguły składni	384
Instrukcje if	384
Ogólny format	384
Proste przykłady	385
Instrukcja wielokrotnego wyboru	386
Instrukcje match	388
Podstawowe użycie match	389
Zaawansowane użycie match	391
Reguły składni Pythona raz jeszcze	393
Ograniczniki bloków — reguły tworzenia wcięć	395
Ograniczniki instrukcji — wiersze i znaki kontynuacji	397
Kilka przypadków specjalnych	398
Testy prawdziwości i testy logiczne	399
Wyrażenie trójargumentowe if/else	401
Podsumowanie rozdziału	403
Sprawdź swoją wiedzę — quiz	403
Sprawdź swoją wiedzę — odpowiedzi	404
13. Pętle while i for	407
Pętle while	407
Ogólny format	408
Przykłady	408
Instrukcje break, continue, pass oraz else w pętli	409
Ogólny format pętli	409
Instrukcja pass	410
Instrukcja continue	411
Instrukcja break	412
Klauzula else pętli	413
Pętle for	415
Ogólny format	415
Przykłady	416
Techniki tworzenia pętli	421
Pętle z licznikami — range	422
Skanowanie sekwencji — while, range, for	423
Przetasowania sekwencji — funkcje range i len	424
Przechodzenie niewyczerpujące — range kontra wycinki	425
Modyfikowanie list — range kontra listy składane	426
Przechodzenie równoległe — zip	427
Generowanie wartości przesunięcia i elementów — enumerate	430

Podsumowanie rozdziału	431
Sprawdź swoją wiedzę — quiz	432
Sprawdź swoją wiedzę — odpowiedzi	432
14. Iteracje i listy składane	435
Iteracje	436
Protokół iteracyjny	437
Wbudowane funkcje iter i next	439
Inne wbudowane typy iterowalne	443
Listy składane	449
Podstawy list składanych	449
Wykorzystywanie list składanych w plikach	450
Rozszerzona składnia list składanych	452
Listy składane — zawieszenie tematu	454
Narzędzia iteracyjne	454
Inne zagadnienia związane z iteracjami	459
Podsumowanie rozdziału	459
Sprawdź swoją wiedzę — quiz	459
Sprawdź swoją wiedzę — odpowiedzi	460
15. Wprowadzenie do dokumentacji	461
Źródła dokumentacji Pythona	461
Komentarze ze znakami #	462
Funkcja dir	462
Notki dokumentacyjne — __doc__	464
PyDoc — funkcja help	468
PyDoc — raporty HTML	472
Nie tylko notki docstrings — pakiet Sphinx	476
Zbiór standardowej dokumentacji	476
Zasoby internetowe	478
Często spotykane problemy programistyczne	478
Podsumowanie rozdziału	480
Sprawdź swoją wiedzę — quiz	481
Sprawdź swoją wiedzę — odpowiedzi	481
Ćwiczenia do części III	482

Część IV. Funkcje i generatory **485**

16. Podstawy funkcji	487
Dlaczego używamy funkcji?	488
Tworzenie funkcji	489
Podstawowe narzędzia funkcji	489
Zaawansowane narzędzia funkcji	490

Ogólne koncepcje związane z funkcjami	491
Instrukcje def	492
Instrukcje return	492
Instrukcja def uruchamiana jest w czasie wykonywania	493
Wyrażenie lambda tworzy funkcje anonimowe	494
Pierwszy przykład: definicje i wywoływanie	495
Definicja	495
Wywołanie	495
Polimorfizm w Pythonie	496
Drugi przykład: przecinające się sekwencje	497
Definicja	497
Wywołania	498
Raz jeszcze o polimorfizmie	499
Zmienne lokalne	500
Podsumowanie rozdziału	500
Sprawdź swoją wiedzę — quiz	501
Sprawdź swoją wiedzę — odpowiedzi	501
17. Zasięgi	503
Podstawy zasięgów w Pythonie	503
Reguły dotyczące zasięgów	505
Rozwiązywanie nazw — reguła LEGB	506
Przykład zasięgu	509
Zasięg wbudowany	511
Instrukcja global	513
Projektowanie programów: minimalizowanie stosowania zmiennych globalnych	515
Projektowanie programów: minimalizacja modyfikacji dokonywanych pomiędzy plikami	516
Inne metody dostępu do zmiennych globalnych	518
Zasięgi a funkcje zagnieżdżone	519
Szczegóły dotyczące zasięgów zagnieżdżonych	519
Przykłady zasięgów zagnieżdżonych	520
Funkcje fabrykujące: domknięcia	521
Instrukcja nonlocal	523
Podstawy instrukcji nonlocal	524
Instrukcja nonlocal w akcji	524
Przypadki graniczne	526
Opcje zachowania stanu	527
Zmienne nonlocal — modyfikowalne, na wywołanie, LEGB	527
Zmienne globalne — modyfikowalne, ale współdzielone	528
Atrybuty funkcji — modyfikowalne, na wywołanie, jawne	529

Klasy — modyfikowalne, na wywołanie, programowanie zorientowane obiektowo	531
A zwięzłą jest...	531
Zakresy i domyślne wartości argumentów	532
Pętle wymagają wartości domyślnych, nie zasięgów	533
Podsumowanie rozdziału	536
Sprawdź swoją wiedzę — quiz	536
Sprawdź swoją wiedzę — odpowiedzi	537
18. Argumenty	540
Podstawy przekazywania argumentów	540
Argumenty a współdzielone referencje	541
Unikanie modyfikacji argumentów mutowalnych	543
Symulowanie parametrów wyjścia i wielu wyników działania	545
Specjalne tryby dopasowywania argumentów	545
Podstawy dopasowywania argumentów	546
Składnia dopasowania argumentów	547
Dopasowywanie argumentów — szczegóły	548
Przykłady ze słowami kluczowymi i wartościami domyślnymi	549
Przykłady dowolnych argumentów	552
Argumenty tylko ze słowami kluczowymi	557
Argumenty tylko pozycyjne	559
Kolejność argumentów — szczegóły techniczne	561
Kolejność w definicji funkcji	561
Kolejność wywoływania	563
Przykład z funkcją obliczającą minimum	564
Pełne rozwiązanie	565
Bonus	567
Puenta	568
Przykład z uogólnionymi funkcjami działającymi na zbiorach	568
Testowanie kodu	569
Przykład z utworzeniem własnej funkcji print	570
Wykorzystywanie argumentów ze słowami kluczowymi	572
Podsumowanie rozdziału	574
Sprawdź swoją wiedzę — quiz	574
Sprawdź swoją wiedzę — odpowiedzi	575
19. Zaawansowane zagadnienia dotyczące funkcji	577
Koncepcje projektowania funkcji	577
Funkcje rekurencyjne	579
Sumowanie z użyciem rekurencji	580
Implementacje alternatywne	581

Pętle a rekurencja	582
Obsługa dowolnych struktur	583
Obiekty funkcji — atrybuty, adnotacje i tym podobne	589
Obiekty „pierwszej klasy”	589
Introspekcja funkcji	590
Atrybuty funkcji	591
Adnotacje funkcji i dekoratory	592
Funkcje anonimowe — lambda	596
Podstawy wyrażeń lambda	596
Po co używamy wyrażeń lambda?	597
Jak (nie) zaciemniać kodu napisanego w Pythonie?	600
Zasięgi: wyrażenia lambda również można zagnieżdżać	601
Narzędzia programowania funkcyjnego	602
Odwzorowywanie funkcji na obiekty iterowalne — map	602
Wybieranie elementów obiektów iterowalnych — funkcja filter	604
Łączenie elementów obiektów iterowalnych — funkcja reduce	605
Podsumowanie rozdziału	606
Sprawdź swoją wiedzę — quiz	606
Sprawdź swoją wiedzę — odpowiedzi	607
20. Listy składane i generatory	610
Listy składane — akt końcowy	610
Powtórka z list składanych	611
Przykład: listy składane i macierze	615
Funkcje i wyrażenia generatorów	618
Funkcje generatorów — yield kontra return	619
Wyrażenia generatorów — obiekty iterowalne spotykają złożenia	625
Różne ciekawostki dotyczące generatorów	632
Przykład: generowanie mieszanych sekwencji	637
Sekwencje mieszające	638
Permutacje: wszystkie możliwe kombinacje	641
Przykład: emulowanie funkcji zip i map	646
Tworzymy własną implementację funkcji map	647
Własna wersja funkcji zip i map z Pythona 2.X	648
Funkcje asynchroniczne — krótka historia	651
Podstawy funkcji asynchronicznych	652
Podsumowanie funkcji asynchronicznych	659
Podsumowanie rozdziału	660
Sprawdź swoją wiedzę — quiz	660
Sprawdź swoją wiedzę — odpowiedzi	660

21. Wprowadzenie do pomiarów wydajności	664
Pomiary wydajności domowymi sposobami	664
Moduł pomiaru czasu — ujęcie pierwsze	665
Moduł pomiaru czasu — ujęcie drugie	666
Skrypt mierzący wydajność	669
Wyniki pomiarów czasu iteracji	671
Inne rozwiązania dla modułu do pomiaru czasu	675
Mierzenie czasu iteracji z wykorzystaniem modułu timeit	677
Podstawowe reguły korzystania z modułu timeit	678
Automatyczne testy wydajnościowe z użyciem modułu timeit	682
Pułapki związane z funkcjami	688
Lokalne nazwy są wykrywane w sposób statyczny	688
Wartości domyślne i obiekty mutowalne	690
Funkcje, które nie zwracają wyników	692
Różne problemy związane z funkcjami	692
Podsumowanie rozdziału	693
Sprawdź swoją wiedzę — quiz	694
Sprawdź swoją wiedzę — odpowiedzi	694
Sprawdź swoją wiedzę — ćwiczenia do części IV	694

Część V. Moduły i pakiety **699**

22. Moduły — wprowadzenie	701
Moduły — podstawy	701
Dlaczego używamy modułów?	702
Architektura programu w Pythonie	703
Struktura programu	703
Importowanie i atrybuty	704
Moduły biblioteki standardowej	706
Jak działa importowanie?	707
1. Odszukanie modułu	707
2. Kompilowanie (o ile jest to potrzebne)	708
3. Wykonanie	710
Ścieżka wyszukiwania modułów	711
Elementy ścieżki wyszukiwania	711
Konfigurowanie ścieżki wyszukiwania	714
Lista sys.path	715
Wybór pliku modułu	717
Nietypowe ścieżki — pliki samodzielne i pakiety	718
Podsumowanie rozdziału	718
Sprawdź swoją wiedzę — quiz	719
Sprawdź swoją wiedzę — odpowiedzi	719

23. Podstawy tworzenia modułów	721
Tworzenie modułów	721
Nazwy modułów	722
Inne rodzaje modułów	722
Używanie modułów	723
Instrukcja import	723
Instrukcja from	723
Instrukcja from *	724
Operacja importowania jest przeprowadzana tylko raz	725
Instrukcje import są przypisaniami	725
Równoważność instrukcji import oraz from	727
Potencjalne pułapki związane z użyciem instrukcji from	728
Przestrzenie nazw modułów	730
Pliki generują przestrzenie nazw	730
Słowniki przestrzeni nazw: <code>__dict__</code>	731
Kwalifikowanie nazw atrybutów	732
Importowanie a zasięgi	733
Zagnieżdżanie przestrzeni nazw	734
Przeładowywanie modułów	736
Podstawy przeładowywania modułów	737
Przykład przeładowywania z użyciem reload	738
Różne aspekty przeładowywania	739
Podsumowanie rozdziału	739
Sprawdź swoją wiedzę — quiz	740
Sprawdź swoją wiedzę — odpowiedzi	740
24. Pakiety modułów	742
Stosowanie pakietów	742
Podstawy importowania pakietów	743
Pakiety a ustawienia ścieżki wyszukiwania	743
Tworzenie pakietów	744
Podstawowa struktura pakietów	744
Pliki <code>__init__.py</code>	747
Pliki <code>__main__.py</code>	748
Do czego służą pakiety?	750
Historia dwóch systemów	750
Role pliku inicjalizacji pakietu	753
Względne importowanie pakietów	755
Względne i bezwzględne importowanie pakietów	755
Importowanie względne — za i przeciw	756
Importy względne w działaniu	757

Pakiety przestrzeni nazw	760
Modele importowania w Pythonie	761
Uzasadnienie dla pakietów przestrzeni nazw	762
Algorytm wyszukiwania modułu	762
Pakiety przestrzeni nazw w akcji	764
Podsumowanie rozdziału	766
Sprawdź swoją wiedzę — quiz	767
Sprawdź swoją wiedzę — odpowiedzi	767
25. Zaawansowane zagadnienia związane z modułami	769
Koncepcje związane z projektowaniem modułów	769
Ukrywanie danych w modułach	771
Minimalizacja niebezpieczeństw użycia <code>from * — _X</code> oraz <code>__all__</code>	771
Zarządzanie dostępem do atrybutów — <code>__getattr__</code> i <code>__dir__</code>	772
Włączanie opcji z przyszłych wersji Pythona: <code>__future__</code>	774
Mieszane tryby użycia — <code>__name__</code> i <code>__main__</code>	775
Przykład: testy jednostkowe z wykorzystaniem atrybutu <code>__name__</code>	776
Rozszerzenie <code>as</code> dla instrukcji <code>import</code> oraz <code>from</code>	778
Introspekcja modułów	779
Przykład: wyświetlanie modułów za pomocą <code>__dict__</code>	780
Importowanie modułów z użyciem nazwy w postaci ciągu znaków	782
Uruchamianie ciągów znaków zawierających kod	783
Bezpośrednie wywołania: dwie opcje	783
Przykład: przechodnie przeładowywanie modułów	784
Pułapki związane z modułami	794
Kolizje nazw modułów: pakiety i importowanie względne w pakietach	794
W kodzie najwyższego poziomu kolejność instrukcji ma znaczenie	794
Instrukcja <code>from</code> kopiuje nazwy, jednak łączy już nie	795
Instrukcja <code>from *</code> może zaciemnić znaczenie zmiennych	796
Funkcja <code>reload</code> może nie mieć wpływu na obiekty importowane za pomocą <code>from</code>	797
Funkcja <code>reload</code> i instrukcja <code>from</code> a testowanie interaktywne	797
Rekurencyjne importowanie za pomocą <code>from</code> może nie działać	799
Podsumowanie rozdziału	800
Sprawdź swoją wiedzę — quiz	800
Sprawdź swoją wiedzę — odpowiedzi	801
Sprawdź swoją wiedzę — ćwiczenia do części V	801

Część VI. Klasy i programowanie zorientowane obiektowo **805**

26. Programowanie zorientowane obiektowo — wprowadzenie	807
Po co używa się klas?	808
Programowanie zorientowane obiektowo z dystansu	809
Wyszukiwanie atrybutów dziedziczonych	810
Klasy a instancje	812
Wywołania metod klasy	813
Tworzenie drzew klas	814
Przeciążanie operatorów	816
Programowanie zorientowane obiektowo oparte jest na ponownym wykorzystaniu kodu	817
Podsumowanie rozdziału	820
Sprawdź swoją wiedzę — quiz	820
Sprawdź swoją wiedzę — odpowiedzi	821
27. Podstawy tworzenia klas	823
Klasy generują wiele obiektów instancji	823
Obiekty klas udostępniają zachowania domyślne	824
Obiekty instancji są rzeczywistymi elementami	825
Pierwszy przykład	825
Klasy dostosowujemy do własnych potrzeb przez dziedziczenie	828
Drugi przykład	829
Klasy są atrybutami w modułach	830
Klasy mogą przechwytywać operatory Pythona	832
Trzeci przykład	833
Najprostsza klasa Pythona na świecie	835
Klasy od kuchni	837
Jeszcze kilka słów o rekordach: klasy kontra słowniki	839
Podsumowanie rozdziału	841
Sprawdź swoją wiedzę — quiz	841
Sprawdź swoją wiedzę — odpowiedzi	842
28. Bardziej realistyczny przykład	844
Krok 1. — tworzenie instancji	845
Tworzenie konstruktorów	845
Testowanie w miarę pracy	847
Wykorzystywanie kodu na dwa sposoby	848
Krok 2. — dodawanie metod	849
Tworzenie kodu metod	851
Krok 3. — przeciążanie operatorów	853
Udostępnienie sposobów wyświetlania	853

Krok 4. — dostosowywanie zachowania za pomocą klas podrzędnych	855
Tworzenie klas podrzędnych	855
Rozszerzanie metod — niepoprawny sposób	856
Rozszerzanie metod — poprawny sposób	857
Polimorfizm w akcji	860
Dziedziczenie, dostosowanie do własnych potrzeb i rozszerzenie	860
Programowanie zorientowane obiektowo — idea	861
Krok 5. — dostosowanie do własnych potrzeb także konstruktorów	862
Programowanie zorientowane obiektowo jest prostsze, niż się wydaje	864
Inne sposoby łączenia klas	864
Krok 6. — wykorzystywanie narzędzi do introspekcji	868
Specjalne atrybuty klas	869
Uniwersalne narzędzie do wyświetlania	870
Atrybuty instancji a atrybuty klas	872
Nazwy w klasach narzędziowych	872
Ostateczna postać naszych klas	873
Krok 7. i ostatni — przechowanie obiektów w bazie danych	875
Obiekty pickle i shelve	876
Przechowywanie obiektów w bazie danych za pomocą shelve	877
Interaktywna eksploracja obiektów shelve	878
Uaktualnianie obiektów w pliku shelve	880
Przyszłe kierunki rozwoju	881
Podsumowanie rozdziału	882
Sprawdź swoją wiedzę — quiz	883
Sprawdź swoją wiedzę — odpowiedzi	883
29. Szczegóły kodowania klas	886
Instrukcja class	886
Ogólna forma	887
Przykład: atrybuty klasy	887
Metody	890
Przykład metody	891
Inne możliwości wywoływania metod	892
Dziedziczenie	892
Tworzenie drzewa atrybutów	892
Szczegóły dziedziczenia	893
Specjalizacja odziedziczonych metod	893
Techniki interfejsów klas	896
Abstrakcyjne klasy nadrzędne	897
Przestrzenie nazw — cała historia	900
Proste nazwy — globalne, o ile nie są przypisane	900
Nazwy atrybutów — przestrzenie nazw obiektów	901

Zen przestrzeni nazw Pythona — przypisanie klasyfikują zmienne	902
Klasy zagnieżdżone — jeszcze kilka słów o regule LEGB	904
Słowniki przestrzeni nazw — przegląd	907
Łączy przestrzeni nazw — przechodzenie w górę drzewa klas	909
Raz jeszcze o notkach dokumentacyjnych	911
Klasy a moduły	913
Podsumowanie rozdziału	913
Sprawdź swoją wiedzę — quiz	914
Sprawdź swoją wiedzę — odpowiedzi	914
30. Przeciążanie operatorów	915
Podstawy	915
Konstruktory i wyrażenia — <code>__init__</code> i <code>__sub__</code>	916
Często spotykane metody przeciążania operatorów	917
Indeksowanie i wycinanie — <code>__getitem__</code> i <code>__setitem__</code>	919
Wycinki	919
Przechwytywanie przypisań elementu	921
Metoda <code>__index__</code> nie służy do indeksowania!	921
Iteracja po indeksie — <code>__getitem__</code>	922
Obiekty iteratorów — <code>__iter__</code> i <code>__next__</code>	923
Iteratory zdefiniowane przez użytkownika	924
Wiele iteracji po jednym obiekcie	927
Alternatywa: metoda <code>__iter__</code> i instrukcja <code>yield</code>	929
Test przynależności — <code>__contains__</code> , <code>__iter__</code> i <code>__getitem__</code>	934
Dostęp do atrybutów — <code>__getattr__</code> i <code>__setattr__</code>	937
Odwołania do atrybutów	937
Przypisywanie wartości i usuwanie atrybutów	938
Inne narzędzia do zarządzania atrybutami	940
Emulowanie prywatności w atrybutach instancji	941
Reprezentacje łańcuchów — <code>__repr__</code> i <code>__str__</code>	942
Po co nam dwie metody wyświetlania?	943
Uwagi dotyczące wyświetlania	944
Dodawanie prawostronne i miejscowa modyfikacja:	
metody <code>__radd__</code> i <code>__iadd__</code>	946
Dodawanie prawostronne	946
Dodawanie w miejscu	950
Wywołania — <code>__call__</code>	951
Interfejsy funkcji i kod oparty na wywołaniach zwrotnych	953
Porównania — <code>__lt__</code> , <code>__gt__</code> i inne	955
Testy logiczne — <code>__bool__</code> i <code>__len__</code>	956
Destrukcja obiektu — <code>__del__</code>	957
Uwagi dotyczące stosowania destruktorów	958

Podsumowanie rozdziału	959
Sprawdź swoją wiedzę — quiz	960
Sprawdź swoją wiedzę — odpowiedzi	960
31. Projektowanie z użyciem klas	962
Python a programowanie zorientowane obiektowo	962
Polimorfizm to interfejsy, a nie sygnatury wywołań	963
Programowanie zorientowane obiektowo i dziedziczenie — związek „jest”	964
Programowanie zorientowane obiektowo i kompozycja — związki typu „ma”	966
Raz jeszcze procesor strumienia danych	968
Programowanie zorientowane obiektowo a delegacja — obiekty „opakowujące”	971
Pseudoprywatne atrybuty klas	973
Przegląd zniekształcania nazw zmiennych	974
Po co używa się atrybutów pseudoprywatnych?	974
Metody są obiektami — z wiązaniem i bez wiązania	977
Metody związane w akcji	978
Klasy są obiektami — uniwersalne fabryki obiektów	981
Do czego służą fabryki?	982
Dziedziczenie wielokrotne i MRO	983
Jak działa dziedziczenie wielokrotne?	985
Jak działa MRO?	986
Rozwiązywanie konfliktów atrybutów	988
Przykład: listowanie atrybutów klas mieszanych	990
Przykład: wyświetlanie atrybutów ze źródłem dziedziczenia	1000
Inne zagadnienia związane z projektowaniem	1004
Podsumowanie rozdziału	1005
Sprawdź swoją wiedzę — quiz	1005
Sprawdź swoją wiedzę — odpowiedzi	1005
32. Zaawansowane zagadnienia związane z klasami	1007
Rozszerzanie typów wbudowanych	1008
Rozszerzanie typów za pomocą osadzania	1008
Rozszerzanie typów za pomocą klas podrzędnych	1009
Model obiektowy Pythona	1012
Klasy są typami, a typy są klasami	1012
Niektóre instancje są równiejsze od innych	1013
Rozgałęzienie dziedziczenia	1014
Różnica między metaklasą a klasą	1016
I jeden object rządzi wszystkim	1016
Zaawansowane narzędzia do obsługi atrybutów	1017
Sloty: deklaracje atrybutów	1017
Właściwości klas: dostęp do atrybutów	1027
Implementacje atrybutów: <code>__getattr__</code> i deskryptory	1030

Metody statyczne oraz metody klasy	1031
Do czego potrzebujemy metod specjalnych?	1031
Metody statyczne	1032
Alternatywy dla metod statycznych	1033
Używanie metod statycznych i metod klas	1034
Zliczanie instancji z użyciem metod statycznych	1036
Zliczanie instancji z metodami klas	1037
Dekoratory i metaklasy	1040
Podstawowe informacje o dekoratorach funkcji	1041
Pierwsze spojrzenie na funkcję dekoratora zdefiniowaną przez użytkownika	1042
Pierwsze spojrzenie na dekoratory klas i metaklasy	1044
Dalsza lektura	1046
Funkcja super	1047
Podstawy funkcji super	1047
Szczegóły dotyczące funkcji super	1048
Posumowanie funkcji super	1056
Pułapki związane z klasami	1057
Modyfikacja atrybutów klas może mieć efekty uboczne	1058
Modyfikowanie mutowalnych atrybutów klas również może mieć efekty uboczne	1059
Dziedziczenie wielokrotne — kolejność ma znaczenie	1060
Zakresy w metodach i klasach	1061
Różne pułapki związane z klasami	1063
Przesadne opakowywanie	1063
Podsumowanie rozdziału	1064
Sprawdź swoją wiedzę — quiz	1064
Sprawdź swoją wiedzę — odpowiedzi	1065
Sprawdź swoją wiedzę — ćwiczenia do części VI	1065

Część VII. Wyjątki **1073**

33. Podstawy wyjątków	1075
Po co używa się wyjątków?	1075
Role wyjątków	1076
Wyjątki w skrócie	1077
Domyślny program obsługi wyjątków	1077
Przechwytywanie wyjątków	1078
Zgłaszanie wyjątków	1080
Wyjątki zdefiniowane przez użytkownika	1081
Działania końcowe	1081

Podsumowanie rozdziału	1083
Sprawdź swoją wiedzę — quiz	1084
Sprawdź swoją wiedzę — odpowiedzi	1084
34. Szczegółowe informacje dotyczące wyjątków	1086
Instrukcja try	1086
Klauzule instrukcji try	1086
Klauzule except i else	1087
Klauzula finally	1094
Połączone klauzule instrukcji try	1097
Instrukcja raise	1101
Zgłaszanie wyjątków	1101
Klauzula except jako punkt zaczepienia	1102
Zakresy widoczności zmiennych i except as	1102
Przekazywanie wyjątków za pomocą raise	1103
Łańcuchy wyjątków — raise from	1104
Instrukcja assert	1106
Przykład: wychwytywanie ograniczeń (ale nie błędów!)	1107
Instrukcja with i menedżery kontekstu	1108
Podstawowe zastosowanie with	1108
Protokół zarządzania kontekstem	1110
Kilka menedżerów kontekstu	1112
Obsługa zakończenia	1112
Podsumowanie rozdziału	1114
Sprawdź swoją wiedzę — quiz	1115
Sprawdź swoją wiedzę — odpowiedzi	1115
35. Obiekty wyjątków	1116
Klasy wyjątków	1117
Tworzenie klas wyjątków	1117
Do czego służą hierarchie wyjątków?	1119
Wbudowane klasy wyjątków	1122
Kategorie wbudowanych wyjątków	1123
Domyślne wyświetlanie oraz stan	1124
Własne sposoby wyświetlania	1126
Własne dane oraz zachowania	1127
Udostępnianie szczegółów wyjątku	1127
Udostępnianie metod wyjątków	1128
Grupy wyjątków — kolejna gwiazda!	1130
Podsumowanie rozdziału	1133
Sprawdź swoją wiedzę — quiz	1133
Sprawdź swoją wiedzę — odpowiedzi	1134

36. Projektowanie z wykorzystaniem wyjątków	1135
Zagnieżdżanie programów obsługi wyjątków	1135
Przykład: zagnieżdżanie przebiegu sterowania	1137
Przykład: zagnieżdżanie składniowe	1138
Zastosowanie wyjątków	1140
Wychodzenie z głęboko zagnieżdżonych pętli: instrukcja go to	1140
Wyjątki nie zawsze są błędami	1141
Funkcje mogą sygnalizować warunki za pomocą raise	1142
Zamykanie plików oraz połączeń z serwerem	1143
Debugowanie z wykorzystaniem zewnętrznych instrukcji try	1143
Testowanie kodu wewnątrz tego samego procesu	1144
Więcej informacji na temat funkcji sys.exc_info	1145
Wyświetlanie błędów i śladów stosu	1146
Wskazówki i pułapki dotyczące projektowania wyjątków	1147
Co powinniśmy opakować w try?	1147
Jak nie przechwytywać zbyt wiele — unikanie pustych except i wyjątków	1148
Jak nie przechwytywać zbyt mało — korzystanie z kategorii opartych na klasach	1150
Podsumowanie podstaw języka Python	1151
Zbiór narzędzi Pythona	1151
Narzędzia programistyczne przeznaczone do większych projektów	1152
Podsumowanie rozdziału	1156
Sprawdź swoją wiedzę — quiz	1157
Sprawdź swoją wiedzę — odpowiedzi	1157
Sprawdź swoją wiedzę — ćwiczenia do części VII	1157

Część VIII. Zagadnienia zaawansowane **1159**

37. Łańcuchy znaków Unicode oraz łańcuchy bajtowe	1161
Podstawy Unicode	1162
Kodowanie znaków	1162
Kodowanie znaków	1164
Wprowadzenie do narzędzi łańcuchów znaków w Pythonie	1167
Obiekt str	1167
Obiekt bytes	1167
Obiekt bytearray	1168
Pliki binarne i tekstowe	1168
Wykorzystanie ciągów znaków	1169
Literały tekstowe i podstawowe właściwości	1170
Konwersje typów ciągów	1171
Kodowanie łańcuchów znaków Unicode w Pythonie	1173
Deklaracje typu kodowania znaków pliku źródłowego	1178

Wykorzystywanie łańcuchów bajtowych	1181
Wywołania metod	1181
Operacje na sekwencjach	1182
Formatowanie	1183
Inne sposoby tworzenia obiektów bytes	1183
Mieszanie typów łańcuchów znaków	1184
Obiekt bytearray	1185
Wykorzystywanie plików tekstowych i binarnych	1188
Podstawy plików tekstowych	1188
Tryby tekstowy i binarny	1189
Pliki tekstowe Unicode	1191
Unicode, obiekt bytes i inne narzędzia łańcuchów znaków	1194
Moduł dopasowywania wzorców re	1194
Moduł danych binarnych struct	1195
Moduł serializacji obiektów pickle i json	1196
Nazwy plików w funkcji open i inne narzędzia dla nazw plików	1197
Zmierz Unicode	1201
Obsługa BOM w Pythonie	1201
Normalizacja Unicode — dokąd zmierza ten standard?	1205
Podsumowanie rozdziału	1208
Sprawdź swoją wiedzę — quiz	1208
Sprawdź swoją wiedzę — odpowiedzi	1209
38. Zarządzane atrybuty	1211
39. Dekoratory	1212
40. Metaklasy i dziedziczenie	1214
Tworzyć metaklasy czy tego nie robić?	1215
Wady funkcji pomocniczych	1215
Metaklasy a dekoratory klas — runda 1.	1218
Model metaklasy	1219
Klasy są instancjami obiektu type	1219
Metaklasy są klasami podrzędnymi klasy type	1220
Instrukcje class wywołują typ	1221
Instrukcje class mogą wybierać typ	1222
Protokół metod metaklas	1223
Tworzenie metaklas	1224
Prosta metaklasa	1224
Dostosowywanie tworzenia do własnych potrzeb oraz inicjalizacja	1225
Pozostałe sposoby tworzenia metaklas	1226
Zarządzanie klasami za pomocą metaklas i dekoratorów	1229

Dziedziczenie — finał	1235
Metaklasy a klasy nadrzędne	1237
Dziedziczenie metaklas	1239
Algorytm dziedziczenia Pythona — prosta wersja	1240
Algorytm dziedziczenia w Pythonie — trudniejsza wersja	1243
Podsumowanie dziedziczenia	1246
Metody metaklas	1247
Metody metaklasy a metody klasy	1248
Przeciążanie operatorów w metodach metaklasy	1249
Metody metaklas a metody instancji	1250
Podsumowanie rozdziału	1252
Sprawdź swoją wiedzę — quiz	1252
Sprawdź swoją wiedzę — odpowiedzi	1252
41. Wszystko, co najlepsze	1254
Fala zmian w Pythonie	1254
Piaskownica Pythona	1256
Zalety Pythona	1257
Końcowe wnioski	1257
Dokąd dalej?	1258
Na bis: wydrukuj swój certyfikat!	1258

Dodatki **1263**

A. Wskazówki dotyczące użytkowania platformy	1265
Korzystanie z Pythona w systemie Windows	1266
Korzystanie z Pythona w systemie macOS	1272
Korzystanie z Pythona w systemie Linux	1277
Korzystanie z Pythona w systemie Android	1280
Korzystanie z Pythona w systemie iOS	1284
Samodzielne programy i pliki wykonywalne	1285
I tak dalej	1288
B. Rozwiązania ćwiczeń podsumowujących poszczególne części książki	1289
Część I. Wprowadzenie	1289
Część II. Typy i operacje	1292
Część III. Instrukcja i składnia	1298
Część IV. Funkcje i generatory	1301
Część V. Moduły i pakiety	1311
Część VI. Klasy i programowanie zorientowane obiektowo	1316
Część VII. Wyjątki	1324
Skorowidz	1332

Pakiety modułów

Dotychczas kiedy importowaliśmy moduły, ładowaliśmy *pliki*. To typowy model użycia modułów i technika, której w początkach naszej kariery programisty Pythona będziemy używać najczęściej. Importowanie modułów to jednak coś więcej, niż dotychczas sugerowałem. Ten rozdział wprowadza **pakiety** modułów — zestaw plików modułów, które zwykle odpowiadają *folderom* (inaczej *katalogom*) na urządzeniu. W tym rozdziale zostały omówione cztery tematy:

- importy pakietów, które podają część ścieżki do pliku w folderze;
- same pakiety, które organizują moduły w folderach;
- względne importy pakietów, które używają kropek w obrębie pakietu, aby ograniczyć wyszukiwanie;
- pakiety przestrzeni nazw, które tworzą pakiet mogący obejmować wiele folderów.

Jak się przekonasz importowanie pakietów zamienia katalog z naszego komputera na kolejną przestrzeń nazw Pythona, z atrybutami odpowiadającymi podkatalogom oraz plikom modułów znajdujących się w tym katalogu. Jak zobaczymy, importowanie pakietów jest czasami wymagane do rozwiązania problemów z operacjami importowania powstających, kiedy na jednym komputerze zainstalowanych jest kilka plików programów o tej samej nazwie.

Pakiety to temat nieco zaawansowany, zatem wielu czytelników może go odłożyć na później, aż zdobędzie doświadczenie z modułami opartymi na plikach. Pakiety oferują łatwy sposób organizacji plików kodu, który unika konfliktów związanych z tą samą nazwą i jest wykorzystywany przez wiele narzędzi z biblioteki standardowej i narzędzi zewnętrznych, z których będziesz korzystać. Chociaż pakiety, podobnie jak wiele innych rzeczy w Pythonie, stały się z czasem złożone, warto poznać podstawy ich zastosowania.

Stosowanie pakietów

Aby załadować elementy w folderze pakietu, należy użyć zwykłych instrukcji importu i narzędzi, które już znasz, ale z podaniem ścieżki nazw, która odzwierciedla ścieżkę zagnieżdżonych folderów. Zacznijmy od oprowadzenia po pakietach.

Podstawy importowania pakietów

Na poziomie podstawowym importowanie pakietów jest całkiem proste — w miejscu, w którym w instrukcji `import` normalnie wstawiamy nazwę pliku, umieszczamy *ścieżkę* nazw rozdzielonych od siebie kropkami. Na przykład to zadziała w instrukcji `import`:

```
import dir1.dir2.mod
```

Tak samo wygląda to w przypadku instrukcji `from`:

```
from dir1.dir2.mod import var
```

I to samo dotyczy już zaimportowanych elementów w wywołaniach funkcji `reload`:

```
from importlib import reload
reload(dir1.dir2.mod)
```

Ścieżka z kropkami w tych instrukcjach ma odpowiadać ścieżce w *systemie plików* na Twoim urządzeniu, prowadzącej do pliku `mod.py` lub innego elementu z nazwą `mod` (jak już wiesz, może to być na przykład plik z kodem bajtowym lub moduł rozszerzający napisany w C). Powyższe instrukcje wskazują zatem, że na naszym komputerze istnieje katalog `dir1`, a w nim podkatalog `dir2` zawierający plik modułu `mod.py` (lub podobny).

Co więcej, takie operacje importowania sugerują, że katalog `dir1` znajduje się w katalogu nadrzędnym `dir0`, dostępnym dla ścieżki wyszukiwania modułów Pythona. Innymi słowy, powyższe instrukcje sugerują, że w systemie plików istnieje struktura przypominająca poniższą (z separatorami w postaci lewych ukośników stosowanych w systemie Windows).

```
dir0/dir1/dir2/mod.py           # Lub mod.pyc, mod.so i tak dalej
dir0\dir1\dir2\mod.py          # To samo w systemie Windows
```

Katalog nadrzędny `dir0` musi być dodany do ścieżki wyszukiwania modułów (o ile nie jest katalogiem głównym dla pliku najwyższego poziomu) — dokładnie tak samo, jakby `dir1` był plikiem modułu.

Mówiąc bardziej formalnie, pierwszy od lewej element ścieżki importu jest nazwą *względną* w ramach ścieżki wyszukiwania `sys.path`, którą mieliśmy okazję poznać w rozdziale 22. Od tego miejsca do końca ścieżki instrukcje `import` z naszego skryptu w jawny sposób określają ścieżki katalogów prowadzących do modułów.

Składnia z kropkami używana w pakietach jest niezależna od platformy, ale jest to również dowód na to, że ścieżki folderów w instrukcjach importu stają się obiektami zagnieżdżonymi: `dir1.dir2.mod` przechodzi przez trzy obiekty modułów po imporcie. Ta składnia wyjaśnia też, dlaczego Python zgłasza błąd dotyczący pliku, który nie jest pakietem, jeśli w instrukcjach importu podasz rozszerzenie `.py` — to jest traktowane jako import pakietu!

Pakiety a ustawienia ścieżki wyszukiwania

Jeżeli korzystamy z tej opcji, należy pamiętać, że ścieżki katalogów w instrukcjach `import` mogą być tylko *zmiennymi* rozdzielonymi *kropkami*. Nie można w tych instrukcjach użyć żadnej składni ścieżek specyficznej dla określonej platformy, takiej jak `C:\dir1`, `/Users/ja/dir1` czy

`../dir1` — takie ścieżki nie będą działały. Zamiast tego składni specyficznej dla platformy należy użyć w ustawieniach ścieżki wyszukiwania modułów i określić w ten sposób nazwę katalogu nadrzędnego zawierającego Twoje pakiety.

W poprzednim przykładzie katalog `dir0` — nazwa katalogu dodawana do ścieżki wyszukiwania modułów — może być dowolnie długą, specyficzną dla platformy ścieżką do katalogu, prowadzącą do katalogu `dir1`. Zamiast używać niepoprawnej instrukcji, takiej jak poniższa:

```
import C:\Users\ja\kody\dir1\dir2\mod          # Błąd — niepoprawna składnia (Windows)
import /Users/ja/kody/dir1/dir2/mod           # To samo, w systemie Unix
```

dodaj katalog `C:\Users\ja\kody` lub `/Users/ja/kody` do zmiennej środowiskowej `PYTHONPATH`, zapisz w pliku `.pth` umieszczonym w strategicznym miejscu lub dodaj ręcznie do ścieżki `sys.path` w samym kodzie, a następnie wpisz w kodzie programu następujące polecenie:

```
import dir1.dir2.mod                            # OK, zmienne i kropki
```

W rezultacie wpisy ze ścieżki wyszukiwania modułów będą zawierały prefiksy katalogów specyficzne dla platformy i prowadzące do nazw znajdujących się po lewej stronie instrukcji `import` lub `from`. Takie instrukcje importu same z siebie dostarczają pozostałą część ścieżki katalogu w sposób neutralny dla platformy.

Jeżeli chodzi o proste importowanie plików, nie musisz dodawać katalogu nadrzędnego do ścieżki wyszukiwania modułów, jeżeli już tam jest. Zgodnie z tym, co pokazywaliśmy w rozdziale 22., będzie to katalog główny (katalog, w którym pracujesz interaktywnie, lub katalog, w którym znajduje się plik najwyższego poziomu uruchomionego programu), wraz ze standardowym katalogiem biblioteki lub katalogiem *site-packages* instalacji pakietów zewnętrznych. Tak czy inaczej, ścieżka wyszukiwania modułów musi zawierać wszystkie katalogi znajdujące się z lewej strony argumentu instrukcji importowania pakietu kodu.

Tworzenie pakietów

Aby utworzyć własne pakiety, musisz umieścić pliki modułów i zagnieżdżone foldery w folderze pakietu. Folder pakietu może, ale nie musi, zawierać plik `__init__.py`, który jest uruchamiany przy pierwszym imporcie, oraz plik `__main__.py`, który jest uruchamiany, gdy uruchamiany jest cały folder pakietu. W kolejnych punktach zobaczysz krok po kroku, jak to wygląda w kodzie.

Podstawowa struktura pakietów

W najprostszej formie pakiety to foldery zawierające zwykle pliki modułów, a być może także zagnieżdżone podfoldery tego samego typu. Utwórzmy jeden jako przykład. Poniżej zostało użyte wcięcie, aby przedstawić zagnieżdżenie folderów, które będziemy wykorzystywać w tym i następujących punktach:

```
dir0/                                           # Katalog w ścieżce wyszukiwania modułów
  dir1/                                         # Katalog główny pakietu dir1
    mod.py                                     # Plik modułu dir1.mod w pakiecie
```

```

dir2/                                     # Zagnieżdżony folder pakietu dir1.dir2
mod.py                                   # Moduł dir1.dir2.mod w pakiecie

```

Te zagnieżdżone foldery można utworzyć w eksploratorach plików lub za pomocą przedstawionych niżej poleceń, działających na większości platform. W systemie Windows zastosuj lewe ukośniki lub użyj znajdującego się wśród materiałów do pobrania dla tego rozdziału folderu, który ma już zbudowaną strukturę:

```

$ mkdir dir1
$ mkdir dir1/dir2                         # W systemie Windows użyj lewych ukośników

```

Należy pamiętać, że nazwy folderów pakietów, podobnie jak nazwy prostych plików modułów, muszą spełniać zasady dotyczące nazw zmiennych, ponieważ po zaimportowaniu stają się zmiennymi. Zjrzyj do rozdziału 11., aby przypomnieć sobie te ograniczenia. W skrócie: używaj liter, cyfr i znaków podkreślenia, a unikaj słów zarezerwowanych.

Teraz dodaj zagnieżdżone pliki modułów wymienione w przykładach 24.1 i 24.2. Ich kod najwyższego poziomu uruchamia się przy pierwszym imporcie i tworzy atrybuty jak zwykle, a ich tytuły wskazują ich ścieżki (dla spójności podano tutaj separatory uniksowe w postaci ukośników).

Przykład 24.1. *dir1/mod.py*

```

var = 'hakować'
print('Wczytywanie dir1.mod')

```

Przykład 24.2. *dir1/dir2/mod.py*

```

var = 'kod'
print('Wczytywanie dir1.dir2.mod')

```

Użycie podstawowych pakietów

Aby użyć naszego pakietu modułów, zaimportuj jego moduły z poziomu REPL lub innego pliku tak, jakbyś to robił dla prostego pliku *.py*, ale użyj kropek, aby w ścieżce wyszukiwania określić ścieżkę poniżej folderu *dir0* — co w REPL oznacza bieżący katalog:

```

$ python3
>>> import dir1.mod                       # Ścieżka pakietu w imporcie
Wczytywanie dir1.mod
>>> dir1.mod.var                           # Uruchomienie kodu modułu
'hakować'

>>> import dir1.dir2.mod                 # Jeszcze bardziej zagnieżdżona ścieżka
Wczytywanie dir1.dir2.mod
>>> dir1.dir2.mod.var                     # Powtórzenie ścieżki, by uzyskać element na końcu
'kod'

```

Podobnie jak w przypadku prostych modułów najwyższego poziomu, kod modułów zagnieżdżonych w pakiecie jest uruchamiany tylko przy pierwszym imporcie:

```

>>> import dir1.mod                       # Nic się nie dzieje, jeśli moduł został już zaimportowany
>>> import dir1.dir2.mod

```

Ogólnie rzecz biorąc, musisz *zaimportować* zagnieżdżony moduł, aby móc używać jego atrybutów, ponieważ moduły odpowiadające folderom nie odwołują się ponownie do systemu plików podczas pobierania atrybutów i zaimportowanie tylko folderu głównego nie wystarczy:

```
$ python3
>>> import dir1                                # Pobiera dir1, ale nie moduły znajdujące się w środku
>>> dir1.dir2.mod.var
AttributeError: module 'dir1' has no attribute 'dir2'
```

```
$ python3
>>> import dir1.dir2
>>> dir1.dir2.mod.var
AttributeError: module 'dir1.dir2' has no attribute 'mod'
```

```
$ python3
>>> import dir1.dir2.mod                       # Podanie pełnej ścieżki do celu
Wczytywanie dir1.dir2.mod
>>> dir1.dir2.mod.var
'kod'
```

Wszystko to działa tak samo w instrukcji `from`, która, jak widzieliśmy, jest w rzeczywistości importem z dodatkowym kopiowaniem nazw. Jednak wciąż musimy podać folder wraz ze ścieżką w instrukcji importu, aby móc uzyskać dostęp do jego zawartości. Ścieżki wymienione w importach są traktowane dosłownie, a nie pobierane z zmiennych:

```
$ python3
>>> from dir1.mod import var                   # Ścieżki pakietów w instrukcji from
Wczytywanie dir1.mod
>>> var                                       # Nie powtarzaj ścieżki do elementu
'hakować'

>>> from dir1.dir2.mod import var
Wczytywanie dir1.dir2.mod
>>> var
'kod'

>>> from dir1 import dir2                     # Ścieżki odczytywane dosłownie
>>> from dir2 import mod                     # A nie z wartości zmiennych
ModuleNotFoundError: No module named 'dir2'
```

Jedną z potencjalnych zalet użycia instrukcji `from` jest to, że nie trzeba *powtarzać* ścieżki pakietu za każdym razem, gdy chcemy użyć elementu z tego pakietu. W przypadku instrukcji `import` zazwyczaj musisz za każdym razem powtórzyć pełną ścieżkę. Trzeba pamiętać, że `from` pozwala na zapisanie ścieżki tylko raz, a pobranej z niej prostej nazwy można używać wszędzie. Jest to szczególnie przydatne, jeśli struktura katalogu pakietu się zmienia (jak to często bywa w oprogramowaniu):

```
>>> import dir1.dir2.mod                       # import wymaga ścieżek
>>> dir1.dir2.mod.var
'kod'

>>> from dir1.dir2.mod import var             # from może skracać ścieżki
>>> var
'kod'
>>> from dir1.dir2 import mod                 # ...i nie trzeba ich powtarzać
```



```

>>> mod.var
'kod'

>>> import dir1.dir2.mod as mod           # Słowo kluczowe "as" też to potrafi
>>> mod.var
'kod'

```

Jak pokazuje ostatni przykład, rozszerzenie `as` wprowadzone w poprzednim rozdziale może skrócić ścieżki w taki sam sposób jak `from` — podobnie jak `as` upraszcza ręczne przypisanie (więcej o `as` w następnym rozdziale).

Pliki `__init__.py`

Jeśli podczas importowania folderu pakietu musisz uruchomić kod inicjujący, umieść go w pliku o nazwie `__init__.py` i zapisz w folderze pakietu. Python automatycznie uruchomi kod z tego pliku przy pierwszym imporcie pakietu w trakcie działania programu (lub sesji REPL). Oto jak ten plik wzbogaca strukturę naszego pakietu:

```

dir0/
  dir1/
    __init__.py           # Uruchamia się przy pierwszym imporcie dir1
    mod.py
  dir2/
    __init__.py         # Uruchamia się przy pierwszym imporcie dir1.dir2
    mod.py

```

Nowe pliki `__init__.py` znajdziesz w przykładach 24.3 i 24.4.

Przykład 24.3. `dir1/__init__.py`

```

var = 'Python'
print('Uruchomiono dir1.__init__.py')

```

Przykład 24.4. `dir1/dir2/__init__.py`

```

var = 3.12
print('Uruchomiono dir1.dir2.__init__.py')

```

Użycie zaktualizowanego pakietu

Te szczególne pliki `__init__.py` są opcjonalne. Ponieważ zapisany w nich kod jest uruchamiany automatycznie przy pierwszym zaimportowaniu katalogu przez program, służą przede wszystkim jako punkty zaczepienia do uruchomienia inicjalizacji wymaganych przez pakiet (stąd skrócone nazwy). W rzeczywistości ich przypisanie służą do inicjalizacji *przestrzeni nazw* odpowiadającej folderowi na Twoim urządzeniu — podobnie jak w przypadku plików tworzą one atrybuty obiektu modułu pakietu:

```

$ python3
>>> import dir1.dir2.mod           # Uruchamia wszystkie pliki __init__.py
Uruchomiono dir1.__init__.py
Uruchomiono dir1.dir2.__init__.py
Wczytywanie dir1.dir2.mod
>>> dir1.var                       # Nazwa w przestrzeni nazw pliku __init__.py

```

```
'Python'
>>> dir1.dir2.var
3.12
>>> dir1.dir2.mod.var           # Nazwa w przestrzeni nazw zagnieżdżonego pliku
mod.py
'kod'
```

Technicznie rzecz biorąc, pakiety z plikami `__init__.py` nazywane są pakietami „regularnymi”, a pakiety bez nich — pakietami „przestrzeni nazw”. Zanim dodaliśmy plik `__init__.py`, przykład był pakietem „przestrzeni nazw” zawierającym jeden folder, a dodanie `__init__.py` uczyniło go „regularnym”. Szczegółami tego rozróżnienia zajmiemy się później, ale poza tym, że pakiety „regularne” mają pierwszeństwo podczas wyszukiwania modułów, różnica jest dziś głównie zjawiskiem historycznym i nie ma znaczenia w większości zastosowań. Oczywiście pliki `__init__.py` mogą robić więcej niż tylko wyświetlać komunikaty. Ich role zostały omówione dalej w tym rozdziale.

Jak wspomniano wcześniej, `reload` działa również ze ścieżkami pakietów, jeśli już zaimportowałeś odpowiednie ścieżki. Kontynuując poprzednią sesję REPL:

```
>>> from importlib import reload
>>> reload(dir1)
Uruchomiono dir1.__init__.py
<module 'dir1' from '/.../PW6W/r24/dir1/__init__.py'>

>>> reload(dir1.dir2)
Uruchomiono dir1.dir2.__init__.py
<module 'dir1.dir2' from '/.../PW6W/r24/dir1/dir2/__init__.py'>

>>> reload(dir1.dir2.mod)
Wczytywanie dir1.dir2.mod
<module 'dir1.dir2.mod' from '/.../ PW6W/r24/dir1/dir2/mod.py'>
```

Na podstawie informacji wyświetlonych na ekranie można zauważyć, że to wywołanie przeładowuje tylko *pojedynczy* moduł na końcu ścieżki. Aby to poprawić, w następnym rozdziale zakodujemy narzędzie do przeładowywania, które po podaniu głównego katalogu pakietu może przeładować wszystkie elementy na ścieżce pakietu, jeden po drugim (zobacz „Przykład: przechodnie przeładowywania modułów” w następnym rozdziale).

Pliki `__main__.py`

Na koniec, jeśli chcesz, aby użytkownicy pakietu mogli uruchomić go tak, jakby był programem lub skryptem, dodaj pliki `__main__.py` do każdego folderu, w którym chcesz obsługiwać ten tryb. Python automatycznie uruchomi te pliki za każdym razem, gdy folder, który je zawiera, zostanie uruchomiony jak program. Oto jak ten ostatni element wzbogaca strukturę naszego pakietu:

```
dir0/
  dir1/
    __main__.py           # Uruchamiany zawsze, kiedy uruchamiany jest dir1
    __init__.py
    mod__.py
  dir2/
```

```
__main__.py
__init__.py
mod.py
```

```
# Uruchamiany zawsze, kiedy uruchamiany jest dir1.dir2
```

Poniżej znajduje się zawartość dodanych właśnie plików `__main__.py` z przykładów 24.5 i 24.6. Są proste, abyśmy mogli skupić się na pakietach:

Przykład 24.5. `dir1/__main__.py`

```
print('Uruchomiono dir1.__main__.py')
```

Przykład 24.6. `dir1/dir2/__main__.py`

```
print('Uruchomiono dir1.dir2.__main__.py')
```

Użycie zaktualizowanego pakietu

Pliki `__main__.py`, znajdujące się w folderze pakietu, są uruchamiane automatycznie dla różnych opcji uruchamiania, w tym bezpośrednio z wiersza poleceń, które wskazują folder zawierający pakiet. W tym trybie pakiet nie musi znajdować się na ścieżce wyszukiwania modułów:

```
$ python3 dir1                                # Uruchamia __main__.py (nie __init__.py)
Uruchomiono dir1.__main__.py
$ python3 dir1/dir2
Uruchomiono dir1.dir2.__main__.py
$ python3 dir1/dir2/mod.py                    # Uruchamia zagnieżdżony mod.py
Wczytywanie dir1.dir2.mod
```

Pliki `__main__.py` w pakietach są również uruchamiane w trybie `python -m`, który, jak już wiesz, lokalizuje element na ścieżce wyszukiwania modułów i uruchamia go jako skrypt najwyższego poziomu. W przeciwieństwie do bezpośredniego polecenia uruchamiania ten tryb uruchamia pełny mechanizm importu pakietów i wszystkie pliki `__init__.py` na ścieżce, ale pakiet musi się na niej znajdować.

```
$ python3 -m dir1                              # Uruchamia zarówno __init__.py, jak
                                                # i __main__.py
Uruchomiono dir1.__init__.py
Uruchomiono dir1.__main__.py

$ python3 -m dir1.dir2                          # I pakiet musi być na ścieżce wyszukiwania
Uruchomiono dir1.__init__.py
Uruchomiono dir1.dir2.__init__.py
Uruchomiono dir1.dir2.__main__.py

$ python3 -m dir1.dir2.mod                      # Uruchamia mod.py i __init__.py modułu
                                                # nadrzędnego
Uruchomiono dir1.__init__.py
Uruchomiono dir1.dir2.__init__.py
Wczytywanie dir1.dir2.mod
```

Jest to podobne do „pakietów” aplikacji w systemie macOS i w smartfonach, które są w rzeczywistości folderami, lecz mogą być uruchamiane jako elementy wykonywalne. Pliki `__main__.py` pełnią różnorodne role, ale często są używane do zapewnienia interfejsu wiersza poleceń dla

pakietu narzędzi. Dzięki temu możesz zaimportować pakiet, aby używać jego narzędzi w innym programie, ale także uruchomić go jako całość, aby korzystać z narzędzi w samodzielnym programie. Pliku tego można też użyć do umieszczania kodu testowego dla pakietu.

Plik `__main__.py` jest również uruchamiany, gdy znajduje się w pliku ZIP na ścieżce wyszukiwania. Jak wspomniano wcześniej, pliki ZIP są traktowane jak zwykły folder podczas wyszukiwania modułu, ale więcej na ten temat znajdziesz w dokumentacji Pythona.

Trzeba wspomnieć, że importy tych samych pakietów w pliku `__main__.py` mogą wymagać użycia składni `from` (której się nauczysz wkrótce), gdy będą uruchamiane tylko przez argument `-m`. Ta składnia zawodzi w przypadku uruchomień bezpośrednich w wierszu poleceń, więc może być konieczne uruchomienie obsługujące przypadek, gdy `__main__.py` wymaga importów z tego samego pakietu. Jak się przekonasz, podanie pełnej ścieżki podczas importu dla kodu, który ma służyć zarówno jako pakiet, jak i program, może rozwiązać ten problem.

Do czego służą pakiety?

Teraz, gdy zobaczyłeś, jak stosować i tworzyć pakiety, możesz się zastanawiać, do czego właściwie są one potrzebne. Wbrew temu, co mogłeś słyszeć, pakiety są całkowicie opcjonalne i raczej są zbyt skomplikowane na początku Twojej przygody z kodowaniem, a nawet później możesz pisać niesamowite programy bez ich użycia.

Ogólnie rzecz biorąc, pakiety są przydatne, ponieważ pomagają zapewnić unikatowość nazw w większych programach i w bibliotekach publikowanych do użytku przez innych. Organizując kod jako folder pakietu, możesz być niemal pewny, że nazwy jego plików nie będą kolidować z nazwami innego oprogramowania na urządzeniach, na których są zapisane. Pełnią tę samą rolę, jeśli chodzi o separację przestrzeni nazw, jaką pełnią moduły oparte na plikach i lokalne zasięgi w funkcjach. Pakiety jednak opierają się na systemie plików: ponieważ odwołania do Twojego pakietu używają nazwy folderu pakietu, istnieje mniejsze ryzyko kolizji związanych z tą samą nazwą.

Ponadto pakiety mogą sprawić, że importy będą bardziej opisowe, ułatwią zadanie śledzenia lokalizacji zmiennych w plikach kodu i uproszą ustawienia ścieżki wyszukiwania modułów. Jedyna sytuacja, w której importowanie pakietów jest *wymagane* do rozwiązania niejasności, pojawia się, kiedy na jednym komputerze zainstalowana jest większa liczba programów z plikami noszącymi te same nazwy. Jest to problem instalacyjny, jednak w praktyce może stać się dotkliwy — szczególnie biorąc pod uwagę tendencję programistów do używania prostych i podobnych nazw plików modułów.

Aby pomóc Ci zrozumieć, dlaczego to wszystko ma znaczenie, zajmiemy się pewnym hipotetycznym scenariuszem wydarzeń.

Historia dwóch systemów

Załóżmy, że programista tworzy w Pythonie program zawierający plik o nazwie `utilities.py` (ze wspólnym kodem narzędziowym), a także plik najwyższego poziomu `main.py` wykorzystywany przez użytkowników do uruchomienia programu. W całym programie pliki wykorzystują

instrukcję `import utilities` do załadowania wspólnego kodu narzędzi. Kiedy program jest dostarczany klientom, stanowi jedno archiwum `.tar` czy `.zip` zawierające wszystkie pliki programu, a po instalacji rozpakowuje wszystkie pliki do jednego katalogu na komputerze docelowym o nazwie `system1`.

```
system1\  
  utilities.py          # Wspólne funkcje i klasy narzędzi  
  main.py              # Uruchamia program  
  other.py             # Importuje utilities w celu załadowania narzędzi
```

Założmy teraz, że drugi programista tworzy inny program z plikami o nazwach `utilities.py` oraz `main.py` i ponownie wykorzystuje instrukcję `import utilities` w całym programie w celu załadowania pliku ze wspólnym kodem. Kiedy drugi system zostanie pobrany i zainstalowany na tym samym komputerze co pierwszy, jego pliki zostaną rozpakowane do nowego katalogu o nazwie `system2` na komputerze klienta, tak by nie nadpisały plików o tych samych nazwach z pierwszego systemu.

```
system2\  
  utilities.py          # Wspólne narzędzia  
  main.py              # Uruchamia program  
  other.py             # Importuje narzędzia
```

Jak na razie nie ma żadnych problemów — oba systemy mogą współistnieć i mogą być wykonywane na tym samym komputerze. Tak naprawdę nie musimy nawet konfigurować ścieżki wyszukiwania modułów, by skorzystać z obu programów — ponieważ Python zawsze najpierw przeszukuje katalog główny (czyli katalog zawierający plik najwyższego poziomu), operacje importowania w plikach obu systemów automatycznie zobaczą wszystkie pliki w katalogu danego systemu. Jeżeli na przykład klikniemy plik `system1/main.py`, wszystkie operacje importowania najpierw będą przeszukiwały katalog `system1`. W podobny sposób po uruchomieniu pliku `system2/main.py` jako pierwszy przeszukany zostanie katalog `system2`. Należy pamiętać, że ustawienia ścieżki wyszukiwania modułów są potrzebne tylko wtedy, gdy importujemy pliki pomiędzy katalogami.

Założmy jednak, że po zainstalowaniu tych dwóch programów na komputerze decydujemy się użyć jakiejś części kodu z każdego z plików `utilities.py` we własnym programie. W końcu jest to wspólny kod narzędzi, a kod napisany w Pythonie z natury służy do ponownego użycia. W takim przypadku możesz użyć poniższych instrukcji w kodzie pliku utworzonego w trzecim katalogu, tak by załadować jeden z dwóch plików z narzędziami.

```
import utilities  
utilities.func('hakować')
```

I teraz zaczynamy dostrzegać problem. Aby taki kod działał, musimy ustawić ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona katalogi zawierające pliki `utilities.py`. Który katalog należy jednak umieścić jako pierwszy — `system1` czy `system2`?

Problemem jest *liniowa* natura ścieżki wyszukiwania, która zawsze jest przeglądana od lewej do prawej strony, więc bez względu na to, jak długo byśmy się nad tym zastanawiali, zawsze otrzymamy plik `utilities.py` z katalogu wymienionego jako pierwszy (bardziej na lewo) w ścieżce wyszukiwania. W takiej postaci nigdy nie będziemy w stanie zaimportować tego pliku z innego katalogu.

Możemy spróbować zmodyfikować `sys.path` w skrypcie przed każdą operacją importowania, ale to dodatkowa operacja, na dodatek bardzo podatna na błędy, a zmiana ustawień zmiennej `PYTHONPATH` przed każdym uruchomieniem programu w Pythonie jest zbyt nużąca i nie pozwala na używanie *obu* wersji w jednym pliku. Domyślne rozwiązanie sprawia, że jesteśmy w kropce.

Problem ten może rozwiązać właśnie importowanie pakietów. Zamiast instalować programy w niezależnych katalogach wymienionych na ścieżce wyszukiwania modułów osobno, możesz spakować je i zainstalować w *podkatalogach* we wspólnym katalogu głównym. Możemy na przykład zorganizować cały kod tego przykładu w postaci hierarchii katalogów, wyglądającej następująco:

```
root\  
  system1\  
    __init__.py  
    utilities.py  
    main.py  
    other.py  
  system2\  
    __init__.py  
    utilities.py  
    main.py  
    other.py  
  mycode\                # Tutaj lub w dowolnym innym miejscu  
    myfile.py            # Tutaj nasz nowy kod
```

W takiej sytuacji do ścieżki wyszukiwania dodajemy tylko wspólny *katalog główny*. Jeżeli wszystkie operacje importowania w naszym kodzie są wykonywane względem wspólnego katalogu głównego, możemy zaimportować plik narzędzi z *dowolnego* programu za pomocą importowania pakietu — nazwa katalogu zawierającego plik sprawia, że ścieżka (i tym samym referencja do modułu) staje się unikalna. Tak naprawdę możemy nawet zaimportować *oba* pliki narzędzi w tym samym module, dopóki wykorzystujemy instrukcję `import` i powtarzamy pełną ścieżkę za każdym razem, gdy odwołujemy się do modułów narzędzi.

```
import system1.utilities                # Importowanie z jednego pakietu  
import system2.utilities                # Importowanie z drugiego pakietu  
  
system1.utilities.function('hakować')  # I użyj nazwy z dowolnego z nich  
system2.utilities.function('kod')      # Lub obu!
```

Nazwa modułu zawierającego plik sprawia, że referencja do modułu staje się unikalna.

Warto zauważyć, że w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from` tylko wtedy, gdy musimy uzyskać dostęp *do tego samego* atrybutu z dwóch lub większej liczby ścieżek. Gdyby nazwa wywoływanej funkcji była w każdej ścieżce inna, można by było użyć instrukcji `from`, co pozwalałoby uniknąć powtarzania pełnej ścieżki za każdym wywołaniem którejś z funkcji, tak jak opisano to wcześniej; do utworzenia unikalnych synonimów nazw możemy również użyć rozszerzenia `as`.

Z technicznego punktu widzenia w tym przypadku podkatalog *mycode* nie musi się znajdować w *katalogu głównym* — dotyczy to jedynie pakietów kodu, z których będziemy importować. Ponieważ jednak nigdy nie wiemy, kiedy nasze własne moduły będą mogły się przydać

innym programom, możemy równie dobrze od razu umieścić je we wspólnym katalogu głównym w celu uniknięcia problemów z konfliktami między takimi samymi nazwami w przyszłości.

Co ważne, jeżeli będziemy konsekwentnie rozpakowywać wszystkie programy napisane w Pythonie w jednym katalogu głównym, tak jak zaprezentowano to powyżej, konfiguracja ścieżki staje się banalnie prosta — wystarczy do niej tylko raz dodać wspólny katalog główny. Ponadto warto zauważyć, że instrukcje importowania z obu oryginalnych systemów działają bez zmian. Ponieważ najpierw przeszukiwane są ich *katalogi domowe*, dodanie wspólnego katalogu do ścieżki wyszukiwania nie ma znaczenia dla kodu z katalogów *system1* oraz *system2*. Nadal można w nich zastosować polecenie `import utilities` i oczekiwać, że odnajdą w ten sposób własne pliki. Jak zobaczysz dalej, jeśli będziesz używać tych plików *również* jako pakietów, możliwe, że będziesz musiał zastosować importy względne w postaci `from .` (a plik *main.py* mógłby mieć nazwę `__main__.py`).

Na koniec pamiętaj, że nawet jeśli nigdy nie utworzysz własnego pakietu, prawdopodobnie będziesz korzystać z narzędzi z biblioteki standardowej i narzędzi zewnętrznych, które to robią. Oto przykładowe pakiety z biblioteki standardowej:

```
from email.message import Message           # Analiza tekstu e-maila/ tworzenie e-maila
from tkinter.filedialog import askopenfilename # Przenośny zestaw narzędzi do tworzenia GUI
from http.server import CGIHTTPRequestHandler # Narzędzia dla serwera (do Python 3.15?)
```

Poprzez umieszczenie swojego kodu w pakiecie takie narzędzia stają się bardziej samodzielne i unikają konfliktów nazw. To, czy zdecydujesz się zrobić to samo ze swoim kodem, zależy od Ciebie, ale w kolejnych punktach będziemy się zagłębiać jeszcze bardziej w ten temat, na wypadek gdybyś się kiedyś zdecydował na tworzenie własnych pakietów.

Role pliku inicjalizacji pakietu

Teraz, gdy znasz już podstawy i sens korzystania z pakietów, przyjrzyjmy się szczegółom związanym z ich użyciem. Pliki `__init__.py` w pierwszych przykładach były proste, ale te pliki mogą zawierać dowolny kod Pythona, tak jak normalne pliki modułów. Ich nazwy są specjalne, ponieważ ich kod jest uruchamiany automatycznie przy pierwszym imporcie katalogu przez program w Pythonie, co służy głównie jako punkt zaczepienia do wykonywania kroków inicjalizacyjnych wymaganych przez pakiet. Jednak te pliki mogą być również całkowicie puste i czasami mają dodatkowe role.

Pliki `__init__.py` służą jako punkty zaczepienia dla działań odbywających się w czasie inicjalizowania pakietów, deklarują katalogi jako pakiety Pythona, generują przestrzenie nazw dla katalogów i implementują zachowanie instrukcji `from *`, kiedy wykorzystuje się je w połączeniu z importowaniem pakietów.

Inicjalizacja pakietów

Za pierwszym razem, gdy Python importuje coś za pomocą katalogu, automatycznie wykonuje cały kod z pliku `__init__.py` tego katalogu. Z tego powodu pliki te są naturalnym miejscem do wstawienia kodu inicjalizującego stan wymagany przez pakiet, który może

na przykład wykorzystać plik inicjalizujący do utworzenia wymaganych plików z danymi czy otwarcia połączenia z bazą danych. Zazwyczaj pliki `__init__.py` nie są przydatne, jeżeli zostaną wykonane bezpośrednio (do tego służy plik `__main__.py`); są one uruchamiane automatycznie przy pierwszym dostępie do pakietu.

Inicjalizacja przestrzeni nazw modułu

Jak już wspomniano wcześniej, w modelu importowania pakietów ścieżki katalogów ze skryptu stają się po zaimportowaniu prawdziwymi ścieżkami zagnieżdżonych obiektów. Jak widać w poprzednim przykładzie, po zaimportowaniu wyrażenie `dir1.dir2.mod` działa i zwraca obiekt modułu, którego przestrzeń nazw zawiera wszystkie zmienne przypisane przez plik `__init__.py` katalogu `dir2`. Takie pliki udostępniają przestrzeń nazw dla obiektów modułów tworzonych dla katalogów, które w przeciwnym razie nie miałyby żadnego skojarzonego pliku modułu.

Deklaracje użyteczności modułu

Jednym z zadań plików `__init__.py` jest również zadeklarowanie, że dany katalog jest pakietem Pythona. Obecność tych plików zapobiega niezamierzonemu ukrywaniu prawdziwych modułów przez podobne nazwy katalogów, które pojawiają się w ścieżce wyszukiwania modułów. Bez tego zabezpieczenia Python mógłby wybrać katalog, który nie ma nic wspólnego z Twoim kodem, tylko dlatego, że pojawia się wcześniej w ścieżce wyszukiwania. Jak zobaczymy później, pakiety przestrzeni nazw w znacznym stopniu redukują tę rolę, ale uzyskują podobny efekt w sposób algorytmiczny, skanując ścieżkę w poszukiwaniu kolejnych plików. Pliki `__init__.py` w pakietach nadal nadają pakietowi wyższy priorytet niż folderowi o tej samej nazwie znajdującemu się w innym miejscu na ścieżce wyszukiwania.

Zachowanie instrukcji `from *`

Jako zaawansowaną opcję możemy wykorzystać listy `__all__` z plików `__init__.py` do zdefiniowania, co jest eksportowane, kiedy katalog importowany jest za pomocą instrukcji `from *`. W pliku `__init__.py` lista `__all__` ma być listą nazw podmodułów, które powinny zostać zaimportowane, kiedy instrukcji `from *` użyjemy na nazwie pakietu (katalogu). Jeżeli lista `__all__` nie zostanie zdefiniowana, instrukcja `from *` nie załaduje automatycznie podmodułów zagnieżdżonych w katalogu. Zamiast tego załaduje tylko zmienne zdefiniowane przez przypisania w pliku `__init__.py` katalogu, w tym wszystkie podmoduły w jawny sposób zaimportowane przez kod tego pliku. Na przykład użycie instrukcji `from submodule import X` w pliku `__init__.py` katalogu sprawia, że zmienna `X` z podmodułu `submodule` będzie dostępna w przestrzeni nazw tego katalogu bez `__all__`.

Później zobaczysz przykład z listą `__all__`, a więcej informacji znajdziesz w rozdziale 25., gdzie między innymi dowiesz się, że służy ona również do deklarowania eksportów `from *` z prostych plików, nie tylko pakietów, i że jest to część szerszego zagadnienia — **ukrywania danych**. Pliki `__init__.py` mogą również pozostać puste. Dzięki temu podczas wyszukiwania importu Twój pakiet będzie miał pierwszeństwo przed pakietami przestrzeni nazw, ale aby zrozumieć, dlaczego to ma znaczenie, musimy przejść dalej.



Uważaj przy scalaniu klas. Nie powinieneś mylić plików `__init__.py` w pakietach z metodami konstruktora klasy `__init__`, które poznamy w następnej części książki. Pierwsze z nich to pliki z kodem uruchamiane, gdy importy po raz pierwszy przechodzą przez folder pakietu podczas działania programu, a drugie są metodami wywoływanymi w celu utworzenia instancji klasy. Oba komponenty mają role inicjujące i nie są obowiązkowe, ale znacznie różnią się od siebie pomimo podobnych nazw.

Względne importowanie pakietów

Dotychczas podczas omawiania zagadnień związanych z importowaniem pakietów koncentrowaliśmy się głównie na importowaniu plików *spoza* pakietu. Wewnątrz pakietu importowanie plików tego samego pakietu może korzystać z tej samej składni z pełnymi ścieżkami, co importowanie plików *spoza* pakietu — i jak zobaczymy, czasami nawet powinno. Pliki pakietów mogą jednak również korzystać ze specjalnych, uproszczonych reguł importowania *wewnątrz pakietu*, gdzie zamiast określania pełnej ścieżki do modułu w pakiecie, można zastosować ścieżkę *względną* wewnątrz pakietu.

Względne i bezwzględne importowanie pakietów

Kod jest prosty. Importy uruchamiane przez pliki używane jako część pakietu mogą korzystać ze specjalnej składni, takiej jak ta poniżej, która działa tylko w plikach używanych jako komponenty pakietu i tylko w instrukcjach `from`, a nie `import`:

```
from . import module           # Import modułu z tego pakietu (tylko)
from .module import name      # Import nazwy z modułu z tego pakietu
from .. import module         # Import modułu równorzędnego z folderu nadrzędnego
from ..module import name     # Import nazwy równorzędnego modułu względem nadrzędnego
                               # modułu tej nazwy
```

Ta składnia nie miałaby sensu w instrukcji `import`, ponieważ ta instrukcja przypisuje moduły do prostych nazw, a nie ścieżek. Jednak w instrukcji `from`, gdy źródło importu zaczyna się od kropek (lub jest tylko kropką), to `import` jest uznawany za *względny* — identyfikuje element względnie wobec folderu samego pakietu, w którym ten element jest zawarty.

Ta nazwa wywodzi się od względnych ścieżek do *plików*, które odnoszą się do pliku względem bieżącego katalogu roboczego, co omówiono w rozdziale 9. Podobnie jak w przypadku nazw plików kropka oznacza bezpośrednio otaczający folder pakietu, a dwie kropki oznaczają folder o poziom wyżej (każda dodatkowa kropka oznacza kolejny poziom wyżej, choć jest to rzadko używane). Tutaj jednak odwołujemy się do elementu względnie wobec pakietu importującego, a nie do pliku z zawartością względnie wobec bieżącego katalogu roboczego.

Co ważne, względne importowanie w obrębie pakietu przeszukuje *tylko* wskazany pakiet. Foldery wymienione w `sys.path` nigdy nie są sprawdzane, jak podczas zwykłego importowania (wstępne sprawdzenie modułów wbudowanych i zamrożonych jest pomijane). Ponadto składnia importu względnego może być używana *tylko* wtedy, gdy plik jest wykorzystywany jako

część pakietu, w przeciwnym razie nie zadziała. Ma to zarówno dobre, jak i złe konsekwencje, które za chwilę omówimy.

Importowanie w plikach wykorzystywanych jako część pakietu może być jednak kodowane bez kropek jak zwykle, zarówno w instrukcji `import`, jak i `from`:

```
import module                # Import modułu z folderu zawartego w liście sys.path
from module import name     # Import nazwy z tego samego modułu
```

Bez wiodących kropek importowanie jest uznawane za **bezwzględne** (ang. *absolute*) — identyfikuje element znajdujący się w folderze wymienionym w `sys.path`. Jest to normalne zachowanie importów w Pythonie i w rzeczywistości nie ma nic „bezwzględnego” w sensie analogii do ścieżek plików (bezwzględne ścieżki plików to pełne ścieżki w systemie plików). Te „bezwzględne” importy są tak naprawdę **względne** (ang. *relative*) wobec wpisu w ścieżce wyszukiwania modułów, ale takie określenie się już przyjęło.

Należy również podkreślić, że importowanie bezwzględne w obrębie pakietu nie sprawdza automatycznie samego pakietu — pomija pakiet i przechodzi bezpośrednio do przeszukiwania folderów z `sys.path`. Jak widzieliśmy, lista `sys.path` może obejmować folder pakietu dzięki bieżącemu katalogowi robocznemu sesji REPL lub folderowi domowemu skryptu najwyższego poziomu oraz ustawionej zmiennej `PYTHONPATH` lub innym ustawieniom. Jednak bezwzględne importowanie samo w sobie nie sprawdza pakietu dla importów uruchamianych w plikach pakietu.

Importowanie względne — za i przeciw

Dlaczego kropki? Krótko mówiąc, importy względne zapewniają, że importy pakietu będą łączyć jego własne moduły. Ponieważ importy względne przeszukują *tylko* sam pakiet, nie ładują przez przypadek niepowiązanego modułu o tej samej nazwie, znajdującego się gdzieś indziej na maszynie hostującej, wymienionego w `sys.path`. To z kolei czyni pakiet bardziej samodzielnym. Bez importów względnych taki niepowiązany moduł mógłby zakłócić działanie kodu pakietu. Dzięki importowaniu względnemu pakiety są mniej zależne od ustawień ścieżki wyszukiwania klienta, których nie da się przewidzieć ani kontrolować.

Minusem jest to, że ten model jest propozycją typu „wszystko albo nic”. W praktyce musisz zdecydować, jaką rolę mają pełnić Twoje pliki: pakietu czy programu. Importy względne zapewniają widoczność samego pakietu i muszą być używane jako pakiet, natomiast importy bezwzględne nie są ograniczone do pakietów, ale nie dają widoczności dla samego pakietu. Ta kombinacja wydaje się sytuacją bez wyjścia, która ogranicza użyteczność kodu.

Zatem jeśli tworzysz bibliotekę narzędzi, może to nie mieć znaczenia, gdyż możesz przyjąć importy względne w całym swoim pakiecie i dodać plik `__main__.py`, aby uruchomić pakiet jako program z przełącznikiem `-m`. Jeśli jednak piszesz bardziej tradycyjną kolekcję kodu, której chcesz używać dowolnie, opcje wydają się ograniczone.

Jednym z prostych rozwiązań tego dylematu jest całkowite *unikanie* importów względnych i używanie pełnych, jednoznacznych i „bezwzględnych” ścieżek importu pakietu w całym kodzie.

Oznacza to użycie importów, które podają ścieżkę do pożądanego elementu, poczynszysy od folderu głównego pakietu i z jego uwzględnieniem:

```
import package.module           # Import modulu z tego pakietu
from package import module      # To samo co from . import module
from package.module import name # To samo co from .module import name
```

Aby to zadziałało, folder główny pakietu musi znajdować się w folderze wymienionym w `sys.path`, ale to będzie *zwykły* przypadek i klienci nie będą miały możliwości zaimportowania kodu pakietu. To również nie wspiera egzotycznych importów bezpośrednich, takich z użyciem dwóch kropek dla katalogów nadrzędnych, ale wydają się one raczej rzadkie (obecnie tylko jedno narzędzie z biblioteki standardowej ich używa).

Dzięki temu rozwiązaniu, które pomija importy względne, foldery z kodem mogą być stosowane bardziej elastycznie, a zarówno bezpośrednie polecenia w wierszu poleceń, jak i przełącznik trybu modułu `-m` mogą być używane do uruchamiania plików w pakiecie jako programów. Następny punkt pokaże, jak to zrobić.

Importy względne w działaniu

Aby pokazać w praktyce teorię omówioną powyżej, wróćmy do prostego pakietu, który utworzyliśmy na początku tego rozdziału. Jak pamiętasz, jego plik `__main__.py` uruchamia się automatycznie, gdy folder jest uruchamiany jako pakiet, ale będzie również pełnił rolę dowolnego pliku najwyższego poziomu w folderze uruchamianym jako skrypt. Kiedy ostatnio widzieliśmy ten plik, zawierał tylko polecenie `print`. Oto przypomnienie, jak wyglądały pliki, których będziemy tutaj używać, abyś nie musiał się cofać:

```
# dir1/mod.py
var = 'hakować'
print('Wczytywanie dir1.mod')

# dir1/__main__.py
print('Uruchomiono dir1.__main__.py')
```

Wystarczy tej teorii, pokażemy zatem kilka prostych przykładów, aby zademonstrować w praktyce koncepcję importów względnych.

Zwykłe importowanie — rozgrzewka

Ten plik `__main__.py` wcześniej działa, ale sprawa się komplikuje, kiedy go uruchamiamy jako skrypt najwyższego poziomu, gdyż wtedy próbuje zaimportować inny moduł znajdujący się w tym samym folderze pakietu, jak w przykładzie 24.7.

Przykład 24.7. `dir1/__main__.py` (zmieniony)

```
import mod
print('Uruchomiono dir1.__main__.py:', dir1.mod.var.upper())
```

Napisany w ten sposób plik może być uruchamiany bezpośrednio z wiersza poleceń (zarówno jako folder, jak i skrypt), ale nie za pomocą przełącznika trybu modułu `-m`, który aktywuje mechanizmy pakietów (zwróć uwagę na dane wyjściowe z `__init__.py` w tym trybie):

```

$ python3 dir1 # Uruchomienie bezpośrednio z wiersza poleceń
Wczytywanie dir1.mod
Wczytywanie dir1.__main__.py: HAKOWAĆ
$ python3 dir1/__main__.py # Uruchomienie jako folder lub plik
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ

$ python3 -m dir1 # Uruchomienie z przełącznikiem trybu modułu
Uruchomiono dir1.__init__.py
ModuleNotFoundError: No module named 'mod'
$ python3 -m dir1.__main__ # Jako główny katalog pakietu lub moduł jawny
Uruchomiono dir1.__init__.py
ModuleNotFoundError: No module named 'mod'

```

Pierwsze dwa polecenia działają, ponieważ plik `__main__.py` jest uruchamiany jako normalny program spoza pakietu i znajduje importowanego „sąsiada”, gdyż w `sys.path` wpisano katalog główny, czyli folder pliku najwyższego poziomu. Problem z ostatnimi dwoma poleceniami polega na tym, że używają one pliku jako części pakietu. Instrukcja `import mod` w pliku `__main__.py` jest wtedy interpretowana jako importowanie *bezwzględne*, co powoduje *pominięcie* otaczającego pakietu. Stąd te błędy.

Importowanie względne — przygoda

Naszą pierwszą reakcją może być dodanie importów *względnych*, aby zadowolić system pakietów, jak w przykładzie 24.8.

Przykład 24.8. `dir1/__main__.py` (zmieniony)

```

from . import mod
print('Uruchomiono dir1.__main__.py:', mod.var.upper())

```

Ten kod naprawia działanie `-m`, ale *psuje* bezpośrednie polecenie w wierszu poleceń:

```

$ python3 -m dir1
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
$ python3 -m dir1.__main__
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ

$ python3 dir1
ImportError: attempted relative import with no known parent package
$ python3 dir1/__main__.py
ImportError: attempted relative import with no known parent package

```

Pierwsze dwa polecenia działają, ponieważ flaga `-m` wywołuje zachowanie pakietu, co umożliwia użycie składni importu względnego (z kropką), która przeszukuje pakiet i znajduje docelowy moduł. Problem z ostatnimi dwoma poleceniami polega na tym, że Python nie pozwala na użycie importów względnych w trybie niebędącym pakietem, co z góry skazuje te uruchomienia na niepowodzenie, niezależnie od ustawień ścieżki wyszukiwania. Stąd te błędy.

Stąd wynika, że musimy *wybrać*, czy nasz kod ma pełnić rolę pakietu, czy nie.

Importowanie bezwzględne — rozwiązanie

Jak już wspomniano, łatwym sposobem na obejście tego ograniczenia jest użycie zwykłych importów pakietowych, które jawnie określają bezwzględne ścieżki importu (które, nawiasem mówiąc, są tak naprawdę względne wobec `sys.path`), jak w przykładzie 24.9.

Przykład 24.9. `dir1/__main__.py` (zmieniony)

```
import dir1.mod
print('Uruchomiono dir1.__main__.py:', dir1.mod.var.upper())
```

Jak również wspomniano wcześniej, katalog główny pakietu, `dir1`, zazwyczaj będzie na `sys.path`, ponieważ jest to jedyny sposób na korzystanie z jego kodu spoza pakietu (klienty muszą importować za pomocą ścieżek, które również zaczynają się od korzenia pakietu `dir1`).

Aby to tutaj pokazać, jawnie dodamy katalog główny pakietu, używając względnej ścieżki z kropką (w kategoriach *systemu plików*) dla bieżącego katalogu (w typowym użyciu zamiast tego może być ścieżka bezwzględna lub folder *site-packages* dla zainstalowanych pakietów). Musimy to ustawić tutaj, ponieważ folder pliku najwyższego poziomu na ścieżce wyszukiwania w trybie bezpośredniego polecenia znajduje się o jeden poziom *poniżej* folderu głównego pakietu, a zarówno tryb bezpośredni, jak i `-m` potrzebują dostępu do katalogu głównego pakietu:

```
$ export PYTHONPATH=. # Lub podobnie poza systemem Unix — zobacz rozdział 22.
$ python3 dir1
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
$ python3 dir1/__main__.py
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ

$ python3 -m dir1 # Oba tryby działają, importy bez kropki
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
$ python3 -m dir1.__main__
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
```

Teraz uruchomienia *zarówno* bezpośrednio w wierszu poleceń, jak i z przełącznikiem `-m` działają, ponieważ nie korzystamy z narzędzia, które ograniczałoby użyteczność kodu tylko do jednego trybu. W rezultacie mamy kod w *trybie dualnym* — może być używany zarówno jako program, jak i pakiet.

Jeśli wolisz nie powtarzać ścieżek importu przy każdym odniesieniu do elementu, oba tryby uruchamiania również działają, jeśli w pliku `__main__.py` umieścimy jedną z poniżej podanych wersji kodu:

```
import dir1.mod as mod
print('Uruchomiono dir1.__main__.py:', mod.var.upper())

from dir1 import mod
print('Uruchomiono dir1.__main__.py:', mod.var.upper())
```

```
from dir1.mod import var
print('Uruchomiono dir1.__main__.py:', var.upper())
```

Możesz również użyć `from *` w tym schemacie, jeśli dodasz `__all__` do pliku inicjalizacyjnego w katalogu głównym pakietu (z zastrzeżeniem wszystkich standardowych ostrzeżeń o wadach `from *` omówionych w rozdziale 23.):

```
# dir1/__init__.py
__all__ = ['mod']

# dir1/__main__.py
from dir1 import *
print('Uruchomiono dir1.__main__.py:', mod.var.upper())
```

Jeżeli jesteś pewien, że Twój kod będzie wykorzystywany jedynie jako część pakietu, możesz używać importów względnych we wszystkich jego plikach, aby uzyskać dostęp do modułów tego samego pakietu. Co więcej, rozwiązanie z użyciem ścieżek bezwzględnych można zastosować *jedynie* do plików, które mają być uruchamiane jako *skrypty* najwyższego poziomu. Inne pliki, które są tylko importowane, mogą używać importów względem pakietu.

W obu przypadkach importy względne mają tę zaletę, że moduł o tej samej nazwie zawarty w `sys.path` nie zastąpi przypadkowo kluczowego modułu w pakiecie. Jednakże importowanie względne ogranicza plik do ról jedynie pakietowych, a jeśli chcesz, aby pliki pakietu wspierały *zarówno* uruchomienie, jak i importowanie, importy względne nie są najlepszą opcją.

Należy również zauważyć, że pomijamy tu pewne szczegóły ze względu na ograniczone miejsce. Na przykład tryb `-m` wyjątkowo sprawdza również bieżący katalog roboczy dla importów bezwzględnych, przez co nie musimy ustawiać zmiennej `PYTHONPATH` dla tego trybu. Ponadto niektóre źródła sugerują, że niepowodzenia importu względnego w skryptach wynikają z ich nazwy `'__main__'`, ale pliki uruchamiane z `-m` mają tę samą nazwę i używają innych protokołów, które są zbyt złożone, aby je tutaj omówić.

Ponieważ importy względem pakietu są zarówno podatne na zmiany, jak i głównie przeznaczone dla programistów tworzących biblioteki kodu na dużą skalę dla innych, zainteresowanych tym tematem odsyłamy do dokumentacji Pythona po więcej szczegółów. Moduły są, delikatnie mówiąc, narzędziami bogatymi w funkcje, ale dobrą wiadomością jest to, że pakiety przestrzeni nazw, temat ostatniego podrozdziału, nie dodają żadnej nowej składni, lecz jedynie pozwalają modułowi obejmować wiele folderów. Przejdźmy dalej, aby zobaczyć, jak to działa.

Pakiety przestrzeni nazw

Teraz, gdy wiesz już wiele o pakietach i importowaniu względnym, pozostała do omówienia jeszcze jedna kwestia związana z pakietami. Ewolucyjna ścieżka modułów w Pythonie ostatecznie doprowadziła do tego, co jest znane jako **pakiety przestrzeni nazw** — pakiety, które mogą składać się z jednego lub więcej folderów znajdujących się w różnych miejscach na ścieżce wyszukiwania modułów.

Chociaż pakiety podzielone na różne foldery są raczej nieczęsto stosowane w praktyce, uogólnienia wprowadzone w celu wsparcia pakietów przestrzeni nazw w *algorytmie* wyszukiwania

importu (procedurze) umożliwiają również tworzenie pakietów bez plików `__init__.py`. Ponieważ ten zrewidowany algorytm to teraz po prostu wyszukiwanie importu, pakiety przestrzeni nazw są trochę przypadkowym tematem.

Aby zrozumieć miejsce pakietów przestrzeni nazw w szerszym kontekście modułów, a także zmiany, które one wprowadziły, potrzebujemy krótkiej lekcji historii.

Modele importowania w Pythonie

Python ma cztery modele importu. Poniżej wymieniono je od najstarszych do najnowszych, z reprezentatywnymi, ale niepełnymi przykładami (jak widzieliśmy, `from` pozwala również na użycie symbolu wieloznacznego `*`, a `reload` przeładowuje dowolny przekazany do niego moduł, lecz to są tylko tematy dodatkowe):

Podstawowe moduły

Pierwszy, oryginalny model importowania: import plików i ich zawartości względem ścieżki wyszukiwania modułów `sys.path`:

```
import moduł
from moduł import nazwa
```

Podstawowe pakiety

Pierwszy, oryginalny model importowania pakietów, który dodaje rozszerzenia ścieżki katalogu względem ścieżki wyszukiwania modułu `sys.path`, gdzie każdy pakiet jest zawarty w jednym katalogu i ma plik inicjujący:

```
import folder.folder.moduł
from folder.moduł import nazwa
```

Import względem pakietu

Model zastosowany do importu wewnątrz pakietów, omawianego w poprzednim podrozdziale, wraz z jego względnymi lub bezwzględnymi schematami wyszukiwania dla importów z użyciem kropek lub bez:

```
from . import moduł
from .moduł import nazwa
```

Pakiety przestrzeni nazw

Najnowszy model pakietu przestrzeni nazw, który jest nadal zależny od `sys.path`, ale pozwala pakietom składać się z wielu katalogów i nie wymaga pliku inicjującego:

```
import dowolny_folder.dowolny_folder.moduł
from dowolny_folder import nazwa
```

Pierwsze dwa modele są całkowicie samowystarczalne, trzeci zastrza kolejność wyszukiwania i rozszerza składnię dla importów wewnątrz pakietu, a czwarty całkowicie odwraca niektóre podstawowe koncepcje i wymagania poprzedniego modelu pakietu. W praktyce spowodowało to, że Python posiadają teraz tylko dwa warianty pakietów:

- Oryginalny model, obecnie znany jako **pakiety regularne** lub po prostu **zwykłe** (ang. *regular packages*).
- Model alternatywny, znany jako **pakiety przestrzeni nazw** (ang. *namespace packages*).

Oryginalne i nowe modele pakietów nie wykluczają się wzajemnie i mogą być używane jednocześnie w tym samym programie. W rzeczywistości nowy model pakietów przestrzeni nazw działa jako *opcja rezerwowa*, używana tylko wtedy, gdy normalne moduły i standardowe pakiety o tej samej nazwie nie są obecne w ścieżce wyszukiwania modułów. Pomimo ich efektywnej nazwy pakiety przestrzeni nazw są w rzeczywistości jedynie modyfikacją wyszukiwania importu, o czym przekonasz się w kolejnych punktach.

Uzasadnienie dla pakietów przestrzeni nazw

Pakiet przestrzeni nazw nie różni się zasadniczo od zwykłego pakietu; to po prostu nieco inny sposób tworzenia pakietów. Co więcej, na najwyższym poziomie nadal są one względne w stosunku do ścieżki `sys.path`: pierwszy komponent ścieżki pakietu przestrzeni nazw (zapisywanej z kropkami) musi nadal znajdować się w normalnej ścieżce wyszukiwania modułów.

Jednak pod względem budowy fizycznej oba rodzaje pakietów mogą się znacznie od siebie różnić. Zwykle pakiety nadal muszą być zapisane w jednym katalogu i posiadać plik `__init__.py`, który jest uruchamiany automatycznie. W przeciwieństwie do nich nowe pakiety przestrzeni nazw nie mogą zawierać pliku `__init__.py` i mogą składać się z wielu katalogów wykorzystywanych podczas importu. W rzeczywistości *żaden* z katalogów tworzących pakiet przestrzeni nazw nie może mieć pliku `__init__.py`, ale zawartość zagnieżdżona w każdym z nich jest traktowana jak pojedynczy pakiet.

Uzasadnienie dla istnienia pakietów przestrzeni nazw jest zakorzenione w celach *instalacji* pakietów, które mogą wydawać się nie do końca oczywiste, chyba że jesteś odpowiedzialny za takie zadania. Krótko mówiąc, pakiety przestrzeni nazw radykalnie rozwiązują problem związany z możliwością wystąpienia kolizji wielu plików `__init__.py` podczas scalania części pakietu poprzez całkowite usunięcie tego pliku. Ponadto dzięki zapewnieniu standardowej obsługi pakietów, które można podzielić na wiele katalogów i umieszczać w kolejnych wpisach w ścieżce `sys.path`, pakiety przestrzeni nazw zwiększają elastyczność instalacji i zapewniają zintegrowany mechanizm zastępujący wiele niekompatybilnych ze sobą rozwiązań, które były używane do tej pory.

Chociaż jest jeszcze zbyt wcześnie, aby ocenić rzeczywistą przydatność pakietów przestrzeni nazw, przeciętni użytkownicy Pythona zwykle określają je jako bardzo użyteczne i alternatywne rozszerzenie zwykłego modelu pakietu — takie, które nie wymaga plików inicjujących i pozwala na użycie dowolnego katalogu kodu jako pakietu do zaimportowania. Aby przekonać się dlaczego, przejdziemy do szczegółów wyszukiwania importów.

Algorytm wyszukiwania modułu

Aby naprawdę zrozumieć, w jaki sposób pakiety przestrzeni nazw zmieniają i rozszerzają wyszukiwanie importów, musimy zajrzeć pod „maskę”, aby zobaczyć, jak działa operacja importowania.

Jak widzieliśmy, Python przeszukuje zestaw potencjalnych folderów. Dla składników importów znajdujących się *najbardziej* z lewej zestawem kandydatów są wszystkie katalogi wymienione

na ścieżce wyszukiwania modułów `sys.path`. Dla składników pakietów, które są zagnieżdżone w importach *pakietów* lub wymienione w importach *względnych*, zestawem kandydatów jest tylko sam pakiet. Podczas gdy foldery pakietów mają tylko jedną instancję danej nazwy, ścieżki wyszukiwania mogą mieć ich wiele.

Sposób, w jaki wyszukiwanie importu wybiera elementy spośród tych kandydatów, jest bardziej subtelny, niż się wydaje. Dla każdego folderu (*nazwa_katalogu*) w zestawie kandydatów wyszukiwania importu Python testuje różne dopasowania do zaimportowanej nazwy w następującej kolejności (używając / dla separatorów folderów i {...} do wskazania wyboru):

1. Jeżeli zostanie znaleziony plik `nazwa_katalogu\nazwa__init__.py`, importowany i zwracany jest zwykły pakiet.
2. Jeżeli zostanie znaleziony plik `nazwa_katalogu\nazwa.{py, pyc}` lub inne rozszerzenie modułu}, importowany i zwracany jest zwykły pakiet.
3. Jeżeli zostanie znaleziony katalog `nazwa_katalogu\nazwa`, zostanie on zapisany, a skanowanie będzie kontynuowane w następnym katalogu ze ścieżki wyszukiwania.
4. Jeżeli żaden z powyższych plików lub katalogów nie zostanie znaleziony, skanowanie będzie kontynuowane od następnego katalogu ze ścieżki wyszukiwania.

Jeżeli skanowanie ścieżki wyszukiwania zakończy się bez zwracania modułu lub pakietu według kroków 1. lub 2., a co najmniej jeden katalog został zapisany w kroku 3., wówczas tworzony jest *pakiet przestrzeni nazw*. W przeciwnym razie zgłaszany jest błąd.

Tworzenie pakietu przestrzeni nazw odbywa się natychmiast i nie jest odraczane do momentu wystąpienia importu na poziomie niżej. Nowy pakiet przestrzeni nazw jest obiektem modułu pozbawionym atrybutu `__file__`, ale ma atrybut `__path__` ustawiony na iterowalną listę ścieżek katalogów, które zostały znalezione i zarejestrowane podczas skanowania w kroku 3.

Atrybut `__path__` jest następnie wykorzystywany w późniejszych, głębszych dostęпах do przeszukiwania wszystkich składników pakietu — każdy katalog w ścieżce `__path__` pakietu przestrzeni nazw jest przeszukiwany, gdy poszukiwane są bardziej zagnieżdżone elementy, podobnie jak to miało miejsce w przypadku katalogu zwykłego pakietu.

Patrząc z innej strony, atrybut `__path__` pakietu przestrzeni nazw pełni tę samą rolę dla komponentów niższego poziomu, co `sys.path` dla elementów ścieżek importu pakietów; staje się „ścieżką nadrzędną” pozwalającą na dostęp do położonych niżej elementów przy użyciu tej samej czteroetapowej procedury, którą omówiliśmy przed chwilą.

W efekcie pakiet przestrzeni nazw jest rodzajem **wirtualnej konkatenacji** katalogów zdefiniowanych za pomocą wpisów w ścieżce wyszukiwania modułów. Jednak po utworzeniu pakietu przestrzeni nazw nie ma funkcjonalnej różnicy między nim a zwykłym pakietem; obsługuje wszystko, czego nauczyliśmy się dla zwykłych pakietów, w tym importowanie względne wewnątrz pakietu.

Co ważne, ponieważ *pojedynczy katalog* pozbawiony pliku `__init__.py`, ale zagnieżdżony w folderze będącym kandydatem do wyszukiwania, przez krok 3. tego algorytmu jest klasyfikowany jako pakiet przestrzeni nazw, każdy taki katalog kwalifikuje się jako pakiet. Jedyną

różnicą między takim pojedynczym folderem a zwykłym folderem pakietu jest to, że ten pierwszy ma niższy *priorytet w wyszukiwaniu*. Zarówno foldery o tej samej nazwie z plikiem `__init__.py`, jak i proste moduły umieszczone w dowolnym miejscu na ścieżce wyszukiwania są wybierane na początku przez kroki 1. i 2.

Innymi słowy, chociaż zmiany wprowadzone w celu wsparcia pakietów przestrzeni nazw sprawiają, że pliki `__init__.py` stają się opcjonalne w folderach pakietów, dodanie takiego pliku, nawet jeśli jest pusty, zapewnia, że pakiet zostanie wybrany zamiast folderu o tej samej nazwie znajdującego się później na ścieżce wyszukiwania (może to również przyspieszyć import, kończąc skanowanie ścieżki wyszukiwania na kroku 1., ale jest to jednorazowe zdarzenie). To, czy ten fakt, czy inne role pliku `__init__.py`, które poznaliśmy wcześniej, uzasadniają dodanie tego pliku, będzie oczywiście zależało od Ciebie i tego, co chcesz osiągnąć.

Pakiety przestrzeni nazw w akcji

Widzieliśmy już, jak działają pakiety przestrzeni nazw w praktyce. Pierwszy krok w przykładzie na początku tego rozdziału utworzył pakiety przestrzeni nazw składające się z *jednego folderu*, ponieważ ich foldery nie miały jeszcze plików `__init__.py`. Ponownie każdy folder zagnieżdżony w folderze na ścieżce wyszukiwania `sys.path` kwalifikuje się jako pakiet, o ile nie jest ukryty przez zwykły pakiet o tej samej nazwie z plikiem `__init__.py` lub prosty moduł znajdujący się gdzieś indziej na ścieżce.

Aby zobaczyć większy efekt łączenia folderów w „wirtualnych” pakietach przestrzeni nazw, przeprowadźmy szybką demonstrację. Na początek utwórz dwa moduły w zagnieżdżonej strukturze katalogów, które mają podkatalogi o nazwie *sub* znajdujące się w różnych katalogach nadrzędnych, *part1* i *part2* (wcięcia pokazują poziom zagnieżdżenia):

```
ns/  
  part1/  
    sub/  
      mod1.py  
  part2/  
    sub/  
      mod2.py
```

Zauważ, że nie ma tutaj plików `__init__.py` — jak wspomniano wcześniej, pliki te nie mogą być używane w pakietach przestrzeni nazw, ponieważ to pliki inicjalizujące są główną różnicą w porównaniu do zwykłych pakietów umieszczonych w jednym folderze. Na potrzeby tego przykładu możemy utworzyć foldery za pomocą takich poleceń konsoli (w systemie Windows zamień `/` na `\` i pomiń `-p` lub użyj eksploratora plików albo, jeśli wolisz, skorzystaj z gotowych folderów znajdujących się w materiałach do pobrania dla tej książki):

```
$ mkdir -p ns/part1/sub          # Dwa podfoldery o tej samej nazwie w różnych folderach  
$ mkdir -p ns/part2/sub        # I podobnie w systemie Windows
```

Moduły na końcu tych ścieżek znajdziesz w przykładach 24.10 i 24.11. Te moduły wyświetlają informacje podczas importu jako ślad.

Przykład 24.10. `ns/part1/sub/mod1.py`

```
print('Wczytywanie ns/part1/sub/mod1')
```

Przykład 24.11. `ns/part2/sub/mod2.py`

```
print('Wczytywanie ns/part2/sub/mod2')
```

Jeżeli dodamy zarówno `part1`, jak i `part2` do ścieżki wyszukiwania modułów, katalog `sub` staje się pakietem przestrzeni nazw obejmującym oba katalogi, z dwoma plikami modułów dostępnymi pod tą nazwą, pomimo że znajdują się one w osobnych katalogach fizycznych. Oto zawartość plików i wymagane ustawienia ścieżek w systemie Unix (w systemie Windows użyj `set i :`, lub zajrzyj do rozdziału 22., gdzie znajdziesz więcej wskazówek). Użyto tutaj ścieżek względem bieżącego katalogu roboczego, ale w praktyce bardziej prawdopodobne byłoby zastosowanie dwóch pełnych ścieżek bezwzględnych:

```
$ export PYTHONPATH=ns/part1:ns/part2
```

Po bezpośrednim zaimportowaniu pakietu przestrzeni nazw staje się *wirtualnym połączeniem* poszczególnych składników katalogu i umożliwia poprzez normalny import dostęp do dalszych zagnieżdżonych części za pomocą jednej, złożonej nazwy (jak zwykle ścieżki zostały skrócone za pomocą wielokropka dla oszczędności miejsca):

```
$ python3
>>> import sub
>>> sub
<module 'sub' (namespace) from
['/.../PW6W/r24/ns/part1/sub', '/.../PW6W/r24/ns/part2/sub']>
# Pakiety przestrzeni nazw: zagnieżdżone ścieżki wyszukiwania

>>> from sub import mod1
Wczytywanie ns/part1/sub/mod1
>>> import sub.mod2
Wczytywanie ns/part2/sub/mod2
# Zawartość z dwóch różnych katalogów

>>> mod1
<module 'sub.mod1' from '/.../PW6W/r24/ns/part1/sub/mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from '/.../PW6W/r24/ns/part2/sub/mod2.py'>
```

Dzieje się tak również wtedy, gdy *natychmiast* importujemy nazwę pakietu za pośrednictwem przestrzeni nazw — ponieważ pakiet przestrzeni nazw jest tworzony przy pierwszym użyciu, czas przeszukiwania ścieżek jest nieistotny:

```
$ python3
>>> import sub.mod1
Wczytywanie ns/part1/sub/mod1
>>> import sub.mod2
Wczytywanie ns/part2/sub/mod2
# Jeden pakiet składający się z dwóch katalogów

>>> sub.mod1
<module 'sub.mod1' from '/.../PW6W/r24/ns/part1/sub/mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from '/.../PW6W/r24/ns/part2/sub/mod2.py'>

>>> sub
<module 'sub' (namespace) from
```

```
[ '../PW6W/Chapter24/ns/part1/sub', '../PW6W/r24/ns/part2/sub']>
>>> sub.__path__
NamespacePath(
[' ../PW6W/r24/ns/part1/sub', '../PW6W/r24/ns/part2/sub'])
```

Co ciekawe, *import względny* działa również w pakietach przestrzeni nazw, tak jak w przykładzie 24.12.

Przykład 24.12. *ns/part1/sub/mod1.py* (zmieniony)

```
from . import mod2
print('Wczytywanie ns/part1/sub/mod1')
```

Dodana instrukcja importu względnego odwołuje się do pliku w pakiecie, mimo że sam plik, do którego się odwołuje, znajduje się w *innym katalogu*:

```
$ python3
>>> import sub.mod1                                # Import względny mod2 z innego folderu
Wczytywanie ns/part2/sub/mod2
Wczytywanie ns/part1/sub/mod1
>>> import sub.mod2                                # Już zaimportowane przez mod1.py — nie jest ponownie
uruchamiany
>>> sub.mod2
<module 'sub.mod2' from '../PW6W/r24/ns/part2/sub/mod2.py'>
```

Jak widać, pakiety przestrzeni nazw są pod każdym względem takie jak zwykłe pakiety z jednym katalogiem, z wyjątkiem dzielonego magazynu fizycznego. To właśnie dlatego foldery z kodem bez pliku *__init__.py* są dokładnie takie same jak zwykłe pakiety, z tym że nie mają kodu inicjującego, który mógłby być uruchamiany automatycznie — są po prostu instancją pakietu przestrzeni nazw z jednym katalogiem.

Jeśli chcesz się dowiedzieć więcej na ten temat, sięgnij po inne źródła dotyczące Pythona. Pakiety przestrzeni nazw są potencjalnie użytecznym narzędziem, ale większość osób uczących się Pythona prawdopodobnie więcej skorzysta na opanowaniu pakietów, które odwzorowują się na rzeczywiste foldery, zanim podejmie się wyzwania, jakim są te pakiety wirtualnie rozprzestrzenione po katalogach.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadziliśmy model *importowania pakietów* Pythona — opcjonalną, lecz przydatną metodę jawnego wymienienia części ścieżki katalogów prowadzących do modułów. Importowanie pakietów nadal odbywa się względem katalogu ze ścieżki wyszukiwania modułów, jednak zamiast pozostawić Pythonowi ręczne przejście ścieżki wyszukiwania, skrypt może podać resztę ścieżki do modułu w sposób jawny. Pakiety mogą być tworzone przy użyciu prostego folderu i mogą sprawić, że importy będą bardziej znaczące, uprościć ustawienia ścieżki wyszukiwania importów oraz rozwiązać niejednoznaczności, gdy istnieje więcej niż jeden moduł o tej samej nazwie — nazwa otaczającego katalogu czyni je unikatowym.

Omówiliśmy również model *importów względnych*, stosowany wyłącznie w pakietach. Jest to sposób wymuszenia importu plików zdefiniowanych w tym samym pakiecie polegający na

zastosowaniu wiodących kropek w ścieżce importu w instrukcji `from`. Na koniec zbadaliśmy *pakiety przestrzeni nazw*, które pozwalają, aby pakiet składał się z wielu katalogów fizycznych; jest to w pewnym sensie „ostatnia deska ratunku” podczas wyszukiwania importów, która całkowicie usunęła wymagania dotyczące plików inicjalizacyjnych w pakietach znajdujących się w jednym folderze.

W kolejnym rozdziale omówimy kilka bardziej zaawansowanych zagadnień związanych z modułami, takich jak użycie atrybutu `__name__` czy `__getattr__`, importowanie nazw oraz narzędzia przydatne podczas pracy z modułami. Jak zawsze jednak zamknijemy bieżący rozdział krótkim quizem sprawdzającym wiadomości w nim przedstawione.

Sprawdź swoją wiedzę — quiz

1. Jaki jest cel umieszczania pliku `__init__.py` w katalogu pakietu modułu?
2. W jaki sposób możemy uniknąć powtarzania pełnej ścieżki pakietu za każdym razem, gdy odnosimy się do zawartości pakietu?
3. Które katalogi wymagają, by znajdował się w nich plik `__init__.py`?
4. Kiedy w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from`?
5. Jaka jest różnica między instrukcją `from pkg import name` a `from . import name`?
6. Czym jest pakiet przestrzeni nazw?

Sprawdź swoją wiedzę — odpowiedzi

1. Plik `__init__.py` służy do deklarowania i inicjalizacji zwykłego pakietu modułu. Python automatycznie wykonuje jego kod za pierwszym razem, gdy w procesie importujemy moduł za pośrednictwem katalogu. Przypisane zmienne pliku stają się atrybutami obiektu modułu utworzonego w pamięci i odpowiadającego temu katalogowi. Nie jest on opcjonalny — nie możemy importować modułów za pomocą składni pakietów, jeżeli katalog nie zawiera tego pliku.
2. Aby bezpośrednio skopiować zmienne z pakietu, należy użyć instrukcji `from` lub rozszerzenia `as` w połączeniu z instrukcją `import`, zastępując ścieżkę krótszym synonimem. W obu przypadkach ścieżka wymieniana jest tylko w jednym miejscu, czyli instrukcji `from` bądź `import`.
3. Podchwytliwe pytanie! We wcześniejszych wersjach Pythona każdy katalog wymieniony w instrukcjach `import` i `from` musiał zawierać plik `__init__.py`, ale teraz nie jest już to obowiązkowe, gdyż wprowadzono udogodnienia dla pakietów przestrzeni nazw w algorytmie wyszukiwania modułów. Jak wspomniano w odpowiedzi 1., te foldery nadal pełnią ważną, choć opcjonalną rolę, takie jak między innymi zwiększenie priorytetu podczas wyszukiwania importu dla folderu.
4. W przypadku pakietów instrukcji `import` musimy użyć w miejsce `from` jedynie wtedy, gdy potrzebujemy uzyskać dostęp do tej *samej zmiennej* zdefiniowanej w więcej niż jednej

ścieżce. Dzięki instrukcji `import` ścieżka sprawia, że referencje stają się unikalne, natomiast instrukcja `from` pozwala na wykorzystywanie tylko jednej wersji danej nazwy zmiennej (o ile do zmiany nazwy nie użyjesz rozszerzenia `as`).

- Instrukcja `from pkg import name` definiuje import *bezwzględny*: pakiet `pkg` jest wyszukiwany z pominięciem katalogu pakietu, w którym występuje ta instrukcja, to znaczy przeszukiwana jest wyłącznie ścieżka `sys.path`. Instrukcja `from . import nazwa` definiuje import *względny*: nazwa `name` jest poszukiwana w odniesieniu do pakietu, w którym występuje ta instrukcja.
- Pakiet przestrzeni nazw* jest rozszerzeniem modelu importu, który reprezentuje pakiet składający się z jednego lub większej liczby katalogów nie zawierających plików `__init__.py`. Gdy Python znajdzie takie katalogi podczas wyszukiwania importu i nie znajdzie wcześniej prostego modułu lub zwykłego pakietu o takiej nazwie, tworzy pakiet przestrzeni nazw, który jest wirtualnym połączeniem wszystkich znalezionych katalogów o nazwie odpowiadającej nazwie żadanego modułu. Dalsze zagnieżdżone komponenty są wyszukiwane we wszystkich katalogach pakietu przestrzeni nazw. W rezultacie powstaje pakiet bardzo podobny do zwykłego pakietu, ale jego zawartość może być podzielona na wiele katalogów.

Skorowidz

A

abstrakcyjne klasy nadrzędne, 897, 914, 968
adnotacje, 491

 funkcji, 189, 592, 607

algorytm

 dziedziczenia, 1240, 1243

 wyszukiwania modułu, 762

analiza skrótowa, 400, 405

AOT, ahead-of-time, 46

aplikacje mobilne, 47

architektura programu, 86, 703

argument self, 815, 821, 838, 842, 843

argumenty

 a współdzielone referencje, 541

 dopasowanie

 pozycyjne, 546

 słowa kluczowe, 546, 548, 550

 wartości domyślne, 546, 548, 550

 dopasowywanie, 546, 548

 formy dopasowania, 547

 funkcji, 491

 wartości domyślne, 532

 z gwiazdkami, 538

 kolejność

 w definicji funkcji, 561

 w wywołaniu funkcji, 563

 mutowalne, 541, 543

 niemutowalne, 541

 przekazywane, 540, 546

 przez nazwę, 546, 557

 przez przypisanie, 540, 545, 590

 przez referencję, 543

 przez wartość, 541

 przez wskaźnik, 541

 według pozycji, 547

 rozpakowywanie, 546, 553

 tylko pozycyjne, 559

 tylko ze słowami kluczowymi, 557

 w nagłówku funkcji, 548, 552

 w wywołaniu funkcji, 547, 548, 553

 zbieranie, 546, 552

 ze słowami kluczowymi, 572, 576, 847

arkusze kalkulacyjne, 47

ASCII, 114, 130, 198, 298, 1161, 1162, 1165,

 1168, 1210

asynchroniczny generator, 658

atak DoS, 106

atrybut, 491, 704

 __bases__, 837, 907, 998

 __class__, 837, 869

 __dict__, 780, 785, 801, 837, 869, 991, 1019

 __doc__, 911

 __name__, 775, 776

 __slots__, 1017

 self, 825

 title, 88

atrybuty

 „wirtualne”, 1022

 dziedziczone, 810

 emulowanie prywatności, 941

 funkcji, 529, 591

 instancji, 827, 840, 842, 872, 892

 klas, 824, 842, 872, 887, 892

 mieszanych, 990

 modyfikowanie, 1058, 1059

 listowanie, 990

 metaklas, 1235

 modułów, 88

 obliczane dynamicznie, 938

 odwołania, 937

 przypisywanie wartości, 938

 pseudoprywatne klas, 873, 973, 974, 1006

 rozwiązywanie konfliktów, 988

 usuwanie, 938

 wbudowane, 867

 wypisywanie, 997

 wyświetlanie ze źródłem dziedziczenia, 1000

 zaawansowane narzędzia, 1017

 zarządzanie dostępem, 1211

automatyczne
testy wydajnościowe, 682
zarządzanie pamięcią, 49
autotest, 776

B

bazy danych, 45, 875
białe znaki, 594
biblioteka
 Chaquopy, 61
 importlib, 87
 pyjnius, 61
 standardowa, 38
 moduły, 706
biblioteki narzędzi, 50
bitowe
 AND, 157
 OR, 157
 XOR, 157
bity, 156
bloki kodu, 395
blokowy łańcuch znaków, block string, 202
błąd RuntimeError, 662
błędy, 95
BOM, Byte Order Marker, 1166, 1201
 w edytorach tekstowych, 1202
 w Pythonie, 1204
Boost.Python, 45
Briefcase, 64
Buildozer, 64

C

CFFI, 45
chmura
 App Engine, 46
 AWS, 46
 Azure, 46
cieniowanie, 693
Cinder, 63
CPython, 60, 61, 66
cx_freeze, 64
cykle, 587
cykliczne struktury danych, 320
Cython, 42, 46, 63
czyszczenie pamięci, garbage collection, 123, 181

D

dane
 binarne, 1161, 1167, 1195
 tekstowe, 1167
debuger
 IDE, 97
 pdb, 97
debugery, 1154
debugowanie, 95–98
definiowanie funkcji, 492, 495
deklaracje typu kodowania, 1178
dekoratory, decorators, 491, 1040, 1212
 funkcji, 143, 592, 853, 1040, 1065
 działanie, 1041
 klas, 1040, 1065, 1218, 1231, 1253
 działanie, 1044
 składnia, 1041
 własne, 1042
dekorowanie automatyczne metod, 1232
delegacja, 866, 867, 942, 971, 973, 1005, 1006
deskryptory, 940, 1030
destrukcja obiektu, 957
destruktory, 958
desygnator typu, 179
DFLR, 984
Django, 44
docstrings, *Patrz* notki dokumentacyjne
dodawanie
 prawostronne, 946
 w miejscu, 950
dokumentacja, 96, 461, 1153
 standardowa, 476
domknięcia, 521, 590
domyślne wartości argumentów, 532
dopasowywanie, 392
 argumentów, 546–548
 wzorców, 1194
DoS, denial of service, 106
dostęp
 do atrybutów, 772, 937, 977, 1027, 1211
 do metod klas, 977
 do zmiennych globalnych, 518
dostosowywanie, 855
 dynamiczne, 736
 klasy, 860
 konstruktora, 862

- drzewo
 - atrybutów, 892
 - dziedziczenia, 1014
 - klas, 810, 814, 826, 909
 - obiektów, 893
- Durus, 46
- dynamiczne dostosowywanie, 736
- dziedziczenie, 807, 825, 828, 829, 835, 884, 892, 964, 1005, 1235, 1246
 - atrybutów, 893
 - diamantowe, diamond inheritance, 829
 - klasowe, 1014
 - metaklas, 1239
 - nieklasowe, 1014
 - pojedyncze, 984
 - przeszukiwanie drzew, 893
 - rozgałęzienie, 1014
 - wielokrotne, multiple inheritance, 814, 984, 990, 1005
 - działanie, 985
 - kolejność, 1060
- dzielenie
 - bez reszty, 151, 152
 - prawdziwe, 151
 - z odcinaniem, 152

F

- fabryka, factory, 981–983
 - obiektów, 848
- faktoryzacja kodu, 851
- FCO, first-class object model, 589
- FIFO, first in, first out, 305, 585
- Flask, 44
- format JSON, 275, 302
- formatowanie łańcuchów znaków
 - f-string, 111, 221, 234, 235
 - metoda format, 222, 227, 230–233
 - string.Template, 240
 - wyrażenia formatujące, 221, 222
 - zaawansowane, 232, 233
 - f-string, 236–239
- formaty wyświetlania liczb, 148
- f-string, 111, 158, 221, 231–239
- funkcja, 487, 501, *Patrz także* metoda
 - __getattr__, 865
 - __import__, 784, 801
 - __iter__, 460
 - abs, 161
 - any, 458
 - await, 331
 - bin, 155, 176
 - bytes, 1170
 - chr, 243, 1162
 - del, 325
 - dict, 228, 457
 - dir, 112, 462, 481, 773, 995, 1181
 - enumerate, 422, 430, 444, 455
 - eval, 155, 176
 - exec, 86, 90, 345, 783
 - extend, 457
 - filter, 445, 446, 455, 604, 607
 - float, 166, 176, 211
 - format, 234
 - generatora, 926
 - getattr, 782
 - getrefcount, 187
 - help, 220, 470
 - dla narzędzi wbudowanych, 469
 - dla własnego kodu, 471
 - hex, 155, 176
 - input, 382
 - int, 153, 160, 176, 211
 - isfloat, 345
 - isinstance, 318
 - iter, 436, 439, 441, 442
 - len, 106, 198, 205, 322, 424–426
 - list, 185, 218, 267, 276, 443, 456, 628, 1168
 - map, 445, 455, 602, 607, 628, 661, 662
 - math.sqrt, 175
 - math.trunc, 153, 176
 - max, 160, 458
 - min, 160, 458
 - next, 439, 442
 - oct, 155, 176
 - open, 90, 129, 131, 291, 292, 305, 538, 1169, 1197, 1209
 - ord, 243, 446, 1162
 - os.getcwd, 76
 - popen, 91
 - pow, 161, 175
 - print, 331, 373–376
 - printf, 227
 - property, 940
 - range, 422, 423, 432, 443
 - modyfikowanie list, 426
 - pomijanie elementów sekwencji, 425
 - przetasowania sekwencji, 424
 - skanowanie sekwencji, 423

read, 90
 reduce, 458, 605, 607
 reload, 87, 736, 737–740, 797
 repr, 149
 reversed, 434, 456
 round, 160, 161, 176
 set, 131, 167
 shelve.open, 878
 sleep, 652
 slot, 909
 sorted, 279, 313, 434, 456
 str, 106, 149, 253, 296, 943, 1170
 subprocess.run, 91
 sum, 160
 super, 859, 1047, 1056, 1245

- algorytm pobierania atrybutu, 1050
- brak wywołań, 1055
- działanie, 1048
- obiekt pośredniczący, 1048
- przeciążanie operatorów, 1055
- uniwersalne wdrożenie, 1051
- zakotwiczenie łańcucha wywołań, 1052

 sys.exc_info, 1144–1146
 sys.exception, 1146
 tester, 641
 time, 665
 tuple, 456, 544
 type, 132, 316, 1012
 własna

- działania na zbiorach, 568
- map, 647, 648
- obliczanie minimum, 564
- print, 570
- zip, 648

 yield, 331
 zip, 265, 422, 428, 444, 455, 662

- przechodzenie równoległe sekwencji, 427
- tworzenie słowników, 429

 funkcje, 487, 501

- adnotacje, 589, 592, 607
- anonimowe, 494, 596
- argumenty, 491, 532
- asynchroniczne, 651, 652, 659
 - używanie, 655
- atomy, 529, 589, 591
- dekoratory, 592
- fabrykujące, 521, 692, 1226
- generatorów, 619, 630, 639
 - działanie, 621
 - protokół iteracyjny, 620
 - stosowanie, 622
 - zawieszanie stanu, 619
- instrukcja def, 492
- instrukcja return, 492
- matematyczne, 141
- narzędzia podstawowe, 489
- narzędzia zaawansowane, 490
- nie zwracające wyników, 692
- obiekty kodu, 591
- pomocnicze, 1215, 1216
- projektowanie, 577
- przekazywane w argumentach, 590
- rekurencyjne, 579, 607, 910
- rozmiar, 578
- spójność, 578
- sprzęganie, 578
- stanowe, 953
- środowisko wykonawcze, 579
- tworzenie, 489, 492–495, 501
- wbudowane, 1245
- wyrażenie lambda, 494
- wywołanie, 495, 498
- z *, 553
- zagnieżdżone, 519
- zwrotne, 608

G

generatory, 120, 436, 661

- funkcje, 618
- podrzedne, 624
- wyrażenia, 625
- zastosowanie, 641, 643

 generowanie

- mieszanych sekwencji, 637
- wyników, 634
- wyników nieokreślonych, 637

 glify, 1167
 gniazda sieciowe, 318
 graficzny interfejs użytkownika, GUI, 44, 82
 granice bloków, 394
 grupy wyjątków, 1130
 GUI, graphical user interface, 44, 82

H

hermetyzacja, 851, 884
hierarchia typów, 316
hierarchia wyjątków, 1119
HPy, 45

I

IDE, Integrated Development Environment, 67, 82, 1154
 PyCharm, 84
 PyDev, 84
 PyScripter, 84
 Spyder, 84
 VSCode, 84
 Wing, 84
IDLE, Integrated Development and Learning Environment, 67, 79, 82
 opcje, 83
ikony plików, 81
implementacje Pythona, 60
importowanie, 58, 704
 modułów, 86
 kompilowanie, 708
 odszukanie modułu, 707
 podstawowe, 761
 wykonanie, 710
 z użyciem nazwy, 782
 opcjonalne rozszerzeń, 774
 pakietów, 743
 względne, 755, 756, 761
 bezwzględne, 755, 759
 podstawowe, 761
 rekurencyjne, 799
indeksowanie, 324, 919, 921
 listy, 249
 łańcuchów znaków, 326
 słowników, 325
informacje o stanie, 846
instalacja, 67
instancje, 811, 812, 821, 823, 1013
 metody, 1250
 nieklasowe, 1014
instrukcja
 as, 1081
 assert, 1075, 1080, 1084, 1106–8, 1115
 async def, 652, 661
 break, 341, 409, 412, 432, 1140

class, 135, 186, 509, 814, 822, 824, 830, 835, 842, 1253
 forma, 887
 wybieranie typu, 1222
 wywołanie typu, 1221
continue, 409, 411
def, 489, 492, 493, 501
elif, 343
except, 1081
from *, 724, 741, 771, 772, 796
from __future__, 774
from, 89, 723, 727, 740, 755, 778, 795, 799
global, 490, 513
go to, 1140
if/elif/else, 124, 332, 336, 341, 384, 401
 domyślna wartość wyboru, 387
 format, 385
 klauzula else, 414
 zagnieżdżanie, 386
import, 705, 723, 729, 740, 743, 767
 rozszerzenie as, 778
 z from __future__, 774
match/case, 388
 format, 389
 test dopasowań, 392
 wielokrotny wybór, 389
 z funkcją strażnika, 393
 zaawansowane użycie, 391
nonlocal, 490, 507, 523–526, 954
pass, 409, 410
print, 55, 373, 383, 943
pwd, 293
raise, 1075, 1080, 1081, 1084, 1101–6, 1103, 1115, 1142
return, 490, 492, 495, 545, 608, 619, 692
try/finally, 344, 347, 387, 1075–7, 1081–1100, 1143
 działanie, 1088
 klauzula else, 1087, 1091
 klauzula except z as, 1102
 klauzula except*, 1130
 klauzula except, 1087, 1090
 klauzula finally, 1094
 klauzule, 1087
 klauzule połączone, 1097
 zewnętrzna, 1143
with, 1075, 1108–14, 1115
 działanie, 1110
 obsługa zakończenia, 1112
 zastosowanie, 1108

- yield, 459, 491, 619, 661, 692, 929, 932
- yield from, 619, 624
- instrukcje, statements, 329–331, 394, 397
 - i polecenia, 101
 - kilkuwierszowe, 74
 - kolejność, 794
 - przypisania, 331, 348, 349
 - wielokrotnego wyboru, 387, 389
 - wielowierszowe
 - mierzenie czasu działania, 680
 - wyrażeń, 371
 - modyfikacje w miejscu, 372
 - zagnieżdżone, 334, 338, 346
 - złożone, 332, 339, 394
- integracja komponentów, 38, 45
- interakcja, 95
- interfejs użytkownika, UI, 44, 82
- interfejsy, 963
 - funkcji, 953
 - klas, 896
- interpolacja ciągów, 234
- interpreter, 54, 57, 66
- introspekcja, 868
 - modułów, 779
- IronPython, 45, 61
- iteracja, 120, 126, 129
 - po indeksie, 922
- iteracje, 436
 - funkcje iter i next, 439
 - jednoprzebiegowe, 632, 662
 - mierzenie czasu działania, 671, 677, 686
 - narzędzia iteracyjne, 454
 - protokół iteracyjny, 437
 - ręczne, 442
 - typy iterowalne, 443
 - wielokrotne, 927
 - z użyciem yield, 932
- iterator, 276, 436, 438
 - async for, 652, 658
 - generatora, 632
 - jednoprzebiegowy, 634
 - obiektu plikowego, 438
 - plików, 294, 296
 - pojedynczy, 447
 - wielokrotny, 447
 - zagnieżdżanie, 445
 - zdefiniowany przez użytkownika, 924
- iterowalne funkcje generatorów, 459

J

- jakość oprogramowania, 37, 39
- jednostki programów, 104
- JIT, just-in-time, 46
- Jupyter Notebook, 46, 85
- Jython, 45, 61

K

- katalog
 - główny, 712
 - Lib\site-packages, 713
- katalogi
 - biblioteki standardowej, 713
 - PYTHONPATH, 713
 - ścieżek plików .pth, 713
 - z kodem źródłowym, 70
- kategorie
 - typów danych, 241
 - typów obiektów, 317
- Kivy, 44
- klasa
 - ArithmeticError, 1123
 - BaseException, 1122
 - Exception, 1090, 1118, 1123
 - LookupError, 1123
 - object, 1012, 1013, 1016
 - OSError, 1123
 - type, 1220
- klasy, 808, 811, 812, 837, 1012, 1013
 - a moduły, 913
 - abstrakcyjne, 897, 899, 914, 968
 - atrybuty specjalne, 869
 - dodawanie metod, 849
 - dostosowanie, 860
 - dziedziczenie, 808, 828, 860
 - hermetyzacja logiki, 851
 - kompozycja, 808
 - metody, 826, 838, 1248
 - modyfikacja atrybutów, 1058
 - modyfikowalne, 531
 - na wywołanie, 531
 - nadrzędne, nadklasy, 811, 822, 828
 - opakowujące, 971
 - ostrzeżenia, 1063
 - podrzędne, podklasy, 811, 855, 1009
 - pośredniczące, 971
 - problemy, 1058

- klasy
 - przechwytywanie operatorów, 832
 - przeciążanie operatorów, 809
 - rozszerzanie, 860
 - składowe, 827
 - sposoby łączenia, 864
 - tworzenie, 825, 845, 855
 - użytkownika, 134
 - wiele instancji, 809
 - właściwości, 1027
 - wyjątków, 1117
 - wbudowane, 1122
 - własne, 1117
 - zagnieżdżone, 904
 - zakresy, 1061
 - zarządzanie, 1229, 1253
- klasyfikacja typów obiektów, 306
- klauzula
 - as, 778, 1102
 - else, 387, 413, 432
 - except*, 1130, 1131, 1132
 - filter, 629
 - filtrująca if, 452
 - finally, 1077, 1084
- klauzule instrukcji try, 1086
- klucze, 259, 272
 - kolejność wstawiania, 268
 - mapowanie wartości, 271
 - słowników, 325
- kod
 - bajtowy, byte code, 42, 56, 57, 60, 66
 - maszynowy, 42, 59, 60, 66
 - źródłowy, 56, 66, 70
- kody znaków, 198
- kolejka typu FIFO, 585
- komentarze, 74, 78, 146, 203, 394, 481
 - ze znakami #, 462
- kompilacja, 56, 708
- kompilator
 - AOT, 46, 62, 63, 85
 - JIT, 46, 62, 65
- komponenty, 45
- kompozycja, 966, 1005, 1006
- kompozyty, composite, 864
- komunikat o błędzie, 75, 343
- konflikt
 - atrybutów, 988
 - nazw zmiennych, 872
- konkatenacja, concatenation, 109, 205, 257
- konsola, 70
- konstrukcja
 - if/elif/else, 384
 - try/except/else, 1115, 1142, 1144
 - try/except/else/finally, 1075
- konstruktor
 - frozenset, 170
 - tworzenie, 845
- konstruktory, 139, 833, 843
 - dostosowanie, 862
- konwersje
 - jawne, 1171
 - kodu znaków, 212
 - łańcuchów znaków, 211
 - typów, 144, 1171
- kooperacyjne wywoływanie metod, 1052
- kopie, 308, 319
- korzystanie z Pythona
 - w systemie Android, 1280
 - w systemie iOS, 1284
 - w systemie Linux, 1277
 - w systemie macOS, 1272
 - w systemie Windows, 1266
- kotwica łańcucha wywołań, 1053
- krotki, tuple, 90, 104, 127, 285, 323
 - konwersje, 288
 - literały, 286
 - metody, 127, 286, 288
 - nazwane, named tuple, 128, 290, 318
 - niemutowalne, 127, 288, 289
 - operacje, 286
 - porównywanie, 312
 - przypisywanie, 325, 349, 352, 417
 - właściwości składni, 287
- kwalifikacja, 88
 - zmiennych, 732

L

- lambda, *Patrz* wyrażenia lambda
- Latin-1, 1165
- LEGB, 506, 527, 904
- liczby, 104, 105, 138, 241, 306
 - całkowite, integer, 105, 139
 - precyzja, 153
 - dziesiętne, 162
 - formaty wyświetlania, 148

- niemutowalne, 187
 - porównywanie, 312
 - separatory, 158
 - stałoprzecinkowe, 105
 - ułamkowe, 164
 - wymierne, 105
 - zespolone, complex number, 105, 139, 140, 154
 - zmiennoprzecinkowe, floating-point number, 139, 344
 - ograniczenia, 165
 - licznik referencji, 179, 188
 - LIFO, last in, first out, 254, 268, 585, 1135
 - lista
 - __all__, 771, 772
 - atrybutów obiektu, 481
 - modułów, 481
 - sys.path, 715
 - zmian w Pythonie, 1255
 - listy, 104, 116, 245, 282
 - filtrowanie, 256
 - indeksowanie, 249
 - iteracje, 255
 - literały, 247
 - metody, 117, 247, 254
 - modyfikacja w miejscu, 250
 - modyfikowanie, 426
 - mutowalność, 116, 245, 258
 - operacje, 247, 248, 258
 - na typach sekwencyjnych, 116
 - specyficzne, 117
 - porównywanie, 312
 - przypisanie, 349, 351
 - rozpakowywanie literałów, 257
 - składane, list comprehension, 119, 255, 256, 426, 449, 460, 610
 - klauzula filtrująca if, 452
 - klauzula for, 453
 - operacje na macierzach, 615
 - składnia, 449, 613
 - składnia rozszerzona, 452
 - stosowanie, 617, 618
 - w nawiasach kwadratowych, 660
 - w nawiasach okrągłych, 660
 - w plikach, 450
 - zagnieżdżone klauzule for, 613
 - zagnieżdżone pętle, 453
 - sortowanie, 252
 - sprawdzanie granic, 117
 - wycinki, 249
 - wywołania metod, 251
 - zagnieżdżanie, 118
 - literały, 103
 - dwójkowe, 157
 - elipsy, 410
 - liczb
 - całkowitych, 139
 - zespolonych, 140
 - zmiennoprzecinkowych, 139
 - list, 247, 257
 - łańcuchów znaków, 196
 - ósemkowe, 139
 - słowników, 261, 265
 - szesnastkowe, 140, 155
 - tekstowe, 1170
 - zaawansowane f-string, 236–239
- ## Ł
- łańcuchy bajtowe, 1181
 - formatowanie, 1183
 - operacje na sekwencjach, 1182
 - wywołania metod, 1181
 - łańcuchy generatorów, 625
 - łańcuchy wyjątków, 1104
 - łańcuchy znaków, strings, 104, 107, 193, 306
 - analiza składniowa, 219
 - apostrofy, 202
 - atrybuty, 112
 - automatyczna konkatencja, 197
 - blokowe, 202
 - działania na sekwencjach, 213
 - format Unicode, *Patrz* Unicode
 - formatowanie
 - f-string, 111, 221, 234–239
 - metoda format, 222, 227, 230–233
 - string.Template, 240
 - wrażenia formatujące, 221, 222
 - zaawansowane, 232, 233
 - indeksowanie, 206, 207, 326
 - konwersja, 211
 - konwersja na obiekty, 301
 - literały, 195, 196, 1170
 - metody, 110, 194, 214, 216, 219, 220
 - mieszanie typów, 1184
 - modyfikowanie, 213, 216
 - narzędzia, 1167, 1194

- łańcuchy znaków, strings
 - niezmiennosc, immutable, 109, 213, 244
 - operacje, 107, 195, 205
 - pomoc, 112
 - porównywanie, 213, 312
 - potrójne cudzysłowy, 202
 - sekwencje ucieczki, 197–201
 - sposoby kodowania, 113
 - surowe, 201
 - typ bytearray, 1168
 - typ bytes, 1167
 - typ str, 1167
 - użycie metody format, 231, 232
 - w apostrofach i cudzysłowach, 196
 - wielowierszowe, 397
 - wycinki, 206, 208, 210, 219, 244
 - wycinki rozszerzone, 209
 - wrażenia formatujące, 224
 - oparte na słowniku, 226
 - zaawansowane, 225

M

- macierz, 118, 615
- macierze rzadkie, 274
- maszyna wirtualna Pythona, 56, 58
- matplotlib, 46
- menedżery kontekstu, 1108, 1112, 1143
 - async with, 652, 658
 - plików, 304
- metafunkcje, 1041
- metaklasy, metaclasses, 1013, 1016, 1040, 1045, 1214, 1231, 1252
 - a klasy nadrzędne, 1237
 - atrybuty, 1235
 - deklaracje, 1236, 1252
 - dekoratory klas, 1218
 - dostosowywanie typów, 1220
 - dziedziczenie, 1239
 - dziedziczenie z klasy type, 1235
 - inicjalizacja, 1225
 - metody, 1247–50
 - model, 1219
 - protokół metod, 1223
 - przeciążanie operatorów, 1249
 - tworzenie, 1215, 1224, 1226
 - zarządzanie klasami, 1229

- metoda, 815, 1065
 - __add__, 833, 947, 960
 - __bool__, 956
 - __call__, 951, 953, 955, 1223
 - __contains__, 934, 936
 - __del__, 957, 772
 - __get__, 940
 - __getattr__, 772, 937
 - __getattribute__, 940, 1030
 - __getitem__, 919, 922, 934, 936, 960
 - __gt__, 955
 - __iadd__, 946, 950, 960
 - __index__, 921
 - __init__, 816, 821, 833, 843–847, 862, 914
 - __iter__, 440, 923–925, 936, 929
 - __len__, 956
 - __lt__, 955
 - __new__, 916, 1221
 - __next__, 437, 439, 460, 623, 924, 927
 - __radd__, 946–950, 960
 - __repr__, 853, 883, 884, 942–945, 960
 - __set__, 940
 - __setattr__, 938, 939
 - __setitem__, 921
 - __str__, 833, 843, 854, 942–945, 960
 - __sub__, 916
- add, 132
- append, 252, 253, 268, 281
- as_completed, 656
- B.decode, 1171, 1172
- bit_length, 157
- close, 294, 295
- copy, 124, 264, 269, 309
- D.pop, 281
- encode, 1164
- extend, 251, 252, 281
- find, 110, 217, 243
- flush, 294
- format, 220, 227–233, 239
- from_float, 166
- gather, 657
- get, 275
- giveRaise, 817
- insert, 251
- isdigit, 343
- items, 126, 263, 271, 276, 313
- iter, 126, 436, 632
- join, 218, 456, 629

- keys, 126, 276, 278
- L.copy, 258
- list, 254, 262
- next, 126, 436
- pop, 251, 254, 255, 264, 268
- popitem, 268
- re.compile, 318
- read, 129
- readline, 295, 437
- readlines, 433, 438
- remove, 132, 251
- replace, 110, 217, 218
- reverse, 254
- rstrip, 299
- S.encode, 1171, 1172
- send, 623, 624
- sort, 252, 253, 381, 456
- split, 219, 300
- str.find, 205
- throw, 624
- update, 264, 269
- values, 126, 263, 276, 278
- write, 296, 378
- metody, 815, 1065
 - abstrakcyjne, 899
 - automatyczne dekorowanie, 1232
 - bez wiązania, 978
 - definiowanie, 891
 - instancji, 1035, 1250
 - klas, 813, 826, 890, 1031, 1035, 1248
 - używanie, 1034
 - zliczanie instancji, 1037, 1038
 - krotek, 127, 286, 288
 - list, 117, 247, 254
 - łańcuchów znaków, 110, 111, 194, 214, 216, 219, 220
 - metaklas, 1247, 1250
 - plików, 129
 - przeciążania operatorów, 835, 917, 918
 - przesłanie, 829
 - rozszerzanie, 856
 - słowników, 124, 126, 261, 263
 - specjalne, 1031, 1033
 - statyczne, 1031, 1032, 1035, 1065
 - używanie, 1034
 - zliczanie instancji, 1036
 - tworzenie kodu, 851
 - typów znakowych, 215
 - typu __X__, 832
 - wyjątków, 1128
 - wyświetlające informacje, 943
 - wywoływanie, 214, 891, 892
 - z wiązaniem, 1006
 - zakresy, 1061
 - zbioru, 168
 - związane, bound, 977–980
 - zwracanie wyników, 835
- MicroPython, 63
- mikrokontrolery, 47
- model
 - FCO, 589
 - metaklasy, 1219
 - obiektowy, 1012
- modele
 - importowania, 761
 - wykonywania, 60
- moduło, 151
- moduł
 - __future__, 774
 - builtins, 161, 505
 - collections, 290
 - copy, 186, 309, 323
 - csv, 303
 - functools, 607
 - io, 305
 - json, 1197
 - locale, 1172
 - logging, 97
 - math, 106, 161
 - os, 305
 - pickle, 301, 302, 323, 876, 1196, 1210
 - pomiaru czasu, 665, 666
 - PyDoc, 468
 - random, 161
 - re, 194, 220, 318, 1168, 1194, 1210
 - shelve, 301, 305, 876
 - skanujący katalogi, 634
 - statistics, 161
 - string, 240
 - struct, 303, 1167, 1195, 1210
 - sys, 187, 373, 1172
 - testów wydajnościowych, 682
 - time, 566, 652, 665
 - timeit, 566, 677, 681, 682
 - tkinter, 576, 609
 - traceback, 1146
 - types, 318

- moduły, 76, 86, 95, 101, 329, 701
 - a klasy, 913
 - algorytm wyszukiwania, 762
 - atrybut `__name__`, 775
 - atrybuty, 88, 721
 - biblioteki standardowej, 706
 - importowanie, 86, 94, 723, 725
 - introspekcja, 779
 - kody źródłowe, 717
 - kolizje nazw, 794
 - ładowanie, 730
 - maksymalizacja spójności, 770
 - minimalizacja niebezpieczeństw, 771
 - minimalizacja sprzężenia, 770
 - modyfikowanie
 - elementów mutowalnych, 726
 - nazw pomiędzy plikami, 727
 - narzędziowe, 141
 - nazwy klas, 830
 - nazwy, 722
 - projektowanie, 769
 - przeładowywanie, 87, 736, 739
 - przechodnie, 784
 - rekurencyjne, 785, 787
 - testowanie, 791
 - przestrzenie nazw, 730
 - rozszerzeń, 722
 - ścieżki wyszukiwania, 711
 - środowisko wykonywania, 770
 - tworzenie, 721, 740
 - ukrywanie danych, 771
 - używanie, 702, 723
 - wbudowane, 711
 - wybór pliku, 717
 - wyświetlanie, 780
 - zasięgi zmiennych, 733
 - MRO, Method Resolution Order, 829, 983, 984, 987–990, 1006, 1048
 - działanie, 986
 - zastosowanie, 1000
 - mutowalność, mutable, 116, 121, 132, 242, 258
- N**
- narzędzia
 - dla nazw plików, 1197
 - do introspekcji, 868
 - do obsługi atrybutów, 1017
 - do wyświetlania, 870
 - do zarządzania atrybutami, 940
 - introspekcji, 591
 - iteracyjne, 436, 446, 454, 460
 - liczbowe, 141
 - łańcuchów znaków, 1167, 1194
 - numeryczne, 160
 - podstawowe funkcji, 489
 - powłoki, shell tools, 40, 44
 - programowania funkcyjnego, 602
 - Pythona, 1151
 - programistyczne, 50, 1152–5
 - rozszerzenia, 1152
 - wbudowane, 50, 1152
 - zaawansowane funkcji, 490
 - nawiasy, 287
 - klamrowe {}, 104, 121, 235, 260, 278, 336, 397
 - kwadratowe [], 104, 230, 246, 260, 338, 397
 - zwykłe (), 333, 397, 398
 - nawracanie, 1077
 - nazewnictwo, 369
 - nazwy
 - argumentów, 846
 - atrybutów, 732, 901
 - bez zapisu kwalifikującego, 900
 - globalne, 510
 - klas, 830
 - lokalne, 505, 510, 688
 - metod, 832
 - modułów, 722, 794
 - plików, 1197
 - proste, 900
 - z zapisem kwalifikującym, 900
 - zmiennych, 367, 872
 - zniekształcanie, 974
 - niemutowalność, immutable, 127, 131, 136, 216, 242, 288, 321, 326
 - normalizacja Unicode, 1205
 - notacja szesnastkowa, 154
 - notatniki Jupyter, 46, 85
 - notki dokumentacyjne, documentation strings, 146, 236, 394, 481, 911, 912
 - `__doc__`, 464
 - literały, 465
 - narzędzie PyDoc, 468
 - priorytety, 467
 - standardy, 467
 - surowe łańcuchy znaków, 465

- wbudowane, 467
- zawartość, 465
- zdefiniowane przez użytkownika, 464

Nuitka, 63

Numba, 46, 62, 66

NumPy, 42

0

obiekt

- bytearray, 1168, 1185–7
- bytes, 1161, 1167, 1170, 1181
 - tworzenie, 1183
- int, 1168
- None, 132, 315, 501
- str, 1161, 1167, 1170, 1181
- type, 132, 316, 1219

obiekty, 101, 178–180, 306

- „opakowujące”, 971
- cykliczne, 320, 587
- funkcji, 589
- generatorów, 626, 632
- identyczność, 187
- instancji, 821, 823, 825
- iteratorów, iterator objects, 440, 923
- iterowalne, iterable objects, 119, 436, 440, 443
 - łączenie elementów, 605
 - odzworowywanie funkcji, 602
 - standardowe, 448
 - wirtualne, 437
 - wybieranie elementów, 604
- klasy, 821, 981
- kodu, 591
- komponentów programów, 318
- metody, 977
- modyfikacje w miejscu, 184
- mutowalne, 690
- niemutowalne, 136, 169
- podstawowe, 135
- przechowywanie, 299, 301–303, 323, 875
- przestrzeni nazw, 848
- reprezentujące gniazda sieciowe, 104
- serializacja, 301
- składane, 635
- skompilowanego kodu, 104
- sprawdzanie tożsamości, 441
- trwałość, 875, 970
- typów, 316

- uaktualnianie w pliku, 880
- uwalnianie, 181
- wbudowane, 102, 104, 306, 317
 - krotki, 104, 127, 285
 - liczby, 104, 105, 138
 - listy, 104, 116, 245
 - łańcuchy znaków, 104, 107, 193
 - pliki, 104, 128, 291
 - słowniki, 104, 120, 258
 - zbiory, 104, 131
- widoków, 276
- wycinków, 919
- wyjątków, 1116
- wyrażeń regularnych, 318
- zagnieżdżanie, 308
- zmiennie, 850

obsługa

- atrybutów, 1017
- bibliotek, 38
- błędów, 342, 344
- liczb zmiennoprzecinkowych, 344
- wyjątków, 96, 387

odchylenie deskryptorów, 1241

odcinanie, 152

odwołania do zasięgów, 533

odzworowania, mappings, 120, 137, 241, 256

ograniczniki instrukcji, 397

OOP, object-oriented programming, 40, 47, 134, 531, 807, 821, 861, 864

opakowywanie

- kodu, 1063

- w try, 1143, 1147

operacje

- na krotkach, 286

- na listach, 247, 248, 258

- na odzworowaniach, 121

- na plikach, 292

- na sekwencjach, 107, 116, 194, 243, 272, 1182

- na słownikach, 260

- uniwersalne, 325

- wbudowane, 867

operator

- %, 222

- &, 400

- *, 205, 257, 553–555

- ** , 105, 266, 553, 554

- //, 152

- @, 143, 1041

- operator
 - \wedge , 400
 - |, 269, 270
 - +, 105, 109, 205, 248, 257
 - ==, 187, 280, 311
 - >, 280
 - await, 143
 - in, 125, 168, 205
 - is, 187, 311, 441
 - lambda, 143
 - logiczny and, 386, 400, 401
 - logiczny not, 400
 - logiczny or, 400
 - przynależności in, 934
- operatory
 - bitowe, 157
 - dzielenia, 151
 - no-ops, 143
 - porównania, 143, 149
 - priorytety, 143
 - przeciążanie, 145, 307, 853, 915
 - wyrażeń, 141, 142
- optymalizacja, 1155
- osadzanie, 1008
 - obiektów, 866

P

- pakiet
 - PyDoc, 468
 - Pygments, 476
 - Sphinx, 476
- pakiety, 718
 - importowanie, 743
 - bezwzględne, 755, 759
 - względne, 755, 756
 - modułów, 742
 - plik inicjalizacji, 753
 - pliki `__init__.py`, 747, 753
 - pliki `__main__.py`, 748
 - przestrzeni nazw, namespace packages, 760, 761, 768
 - działanie, 764
 - regularne, regular packages, 761
 - stosowanie, 742
 - struktura, 744
 - tworzenie, 744
 - ustawienia ścieżki wyszukiwania, 743

- użycie, 745
- zastosowanie, 750
- pandas, 46
- parsowanie, 207
- permutacje, 641
- pętla
 - for/else, 126, 127, 257, 270, *Patrz także* listy składane
 - format, 415
 - funkcja range, 422
 - protokół iteracji, 460
 - przypisanie krotek, 417
 - rozszerzone przypisanie sekwencji, 418
 - wewnątrz funkcji, 497
 - zagnieżdżanie, 420
 - zastosowanie, 416, 421–31
 - while/else, 335, 340, 407
 - format, 408, 409
 - instrukcja break, 412
 - instrukcja continue, 411
 - instrukcja pass, 410
 - klauzula else, 413
 - literał elipsy, 410
 - przypisanie nazwane, 412
 - zagnieżdżanie, 411
- pętle, 533
 - interaktywne, 340
 - w wyrażeniu lambda, 600
 - z licznikami, 422
 - zdarzeń, 652
- piaskownica, sandbox, 1256
- platforma, framework, 819
- plik
 - `__init__.py`, 747, 753, 767
 - `__main__.py`, 748
- pliki, 104, 128, 291, 306, 326
 - .pth, 713, 719
 - .py, 55, 78, 720
 - .pyc, 56, 720
 - binarne, 130, 297, 1168, 1188, 1190
 - binarne zamrożone, frozen binaries, 64
 - buforowanie, 294
 - deskryptorów, 305
 - generowanie wyników, 634
 - iteratory, 294, 296
 - JSON, 302, 303
 - łańcuchy znaków, 294
 - menedżery kontekstu, 304

- metody, 129, 299
- modułów, 77, 94, 516
- najwyższego poziomu, 78
- narzędzia dla nazw, 1197
- operacje, 292
- otwieranie, 292
- przechowywanie obiektów, 299
 - samodzielne, 718
 - shelve, 880, 883
 - skanowanie, 433
 - tekstowe, 297, 1168, 1188, 1189
 - tekstowe Unicode, 130, 1191
 - w trybie binarnym, 1169, 1188, 1189, 1209
 - w trybie tekstowym, 1168, 1188, 1189, 1209
 - wykonywalne, 64, 1285
 - źródłowe, 76
- podpowiedzi typów, 133, 189
- podwyrażenia, 144
- polimorfizm, 109, 137, 145, 188, 496, 499, 817, 860, 963
- pomiar
 - czasu, 666, 675
 - iteracji, 671, 677, 686
 - sortowania, 681
 - wielu wersji Pythona, 686
 - instrukcji wielowierszowych, 680
 - użycie modułu time, 665
 - użycie modułu timeit, 677
 - wydajności, 664, 669
- pomoc dla narzędzi wbudowanych, 469
- ponowne użycie kodu, 817, 821
- porównywanie, 149, 955
 - łączone, 150
 - rekurencyjne, 311
 - słowników, 313
 - typów mieszanych, 312, 313
- porty szeregowe, 47
- porządek MRO, 1006
- POSIX, Portable Operating System Interface, 44
- PowerShell, 70
- powłoka, 40, 70
- powtórzenia, repetition, 109, 205, 319
- precyzja liczb
 - całkowitych, 153
 - dziesiętnych, 163
 - zmiennoprzecinkowych, 163
- priorytety operatorów, operator precedence, 143
- problemy programistyczne, 478
- procedury, 489
- procesor strumienia danych, 968
- profilowanie, 1154
- programowanie
 - funkcyjne, 48, 521, 602
 - gier, 47
 - numeryczne i naukowe, 46
 - przez dostosowanie, 819
 - systemowe, 44
 - zorientowane obiektowo, OOP, 40, 47, 134, 531, 807, 821, 861, 864
 - delegacja, 971
 - dziedziczenie, 962, 964
 - hermetyzacja, 962
 - kompozycja, 966
 - polimorfizm, 962
 - zalety, 1070
- programy, 77, 95, 101, 329
- projektowanie
 - modułów, 769
 - z użyciem klas, 962
 - z wykorzystaniem wyjątków, 1135
- protokół
 - iteracji, 126, 129, 168, 437, 440, 444, 460
 - metod metaklas, 1223
 - zarządzania kontekstem, 1110
- prototypowanie, 46
 - inkrementalne, 847
- przechwytywanie
 - przypisać elementu, 921
 - wbudowanych metod, 915
 - wbudowanych wyjątków, 1093
 - wyjątków, 1078
- przeciążanie
 - drzewa, 829
 - operatora `__iter__`, 459
 - operatorów, 145, 307, 809, 816, 832, 834, 835, 843, 853, 915, 960, 1249
 - wywołań, 1227
- przedawnienie, deprecation, 369
- przekazywanie argumentów, *Patrz* argumenty
- przekierowanie strumienia, stream redirection, 80
- przenośność programów, 38, 48
- przesłanianie metod, 829
- przestrzeń nazw, namespace, 88, 503, 720, 842, 900, 901
 - łącza, 909
 - moduły, 720, 730
 - słowniki, 907

przetwarzanie obrazu, 47
przypisania, 178, 319, 348, 540, 545, 901, 1244
 do indeksu, 250
 do wielu celów, 349, 350
 do wycinka, 250
krotek, 127, 325, 349, 352, 417
listy, 349, 351
nazwane, 349, 364, 412, 509, 636
 stosowanie, 365
niejawne, 349
podstawowe, 349, 351
rozpakowujące krotki i listy, 349
rozszerzone, 349, 350, 361
rozszerzone sekwencji, 418
sekwencji, 349–352
 wzorce zaawansowane, 352
 w odrębnych wierszach, 381
 z wieloma celami, 360, 381
punkty
 przerwania, breakpoints, 83
 zaczepienia, 747, 753, 809, 832, 890
puste wiersze, 394
PVM, Python Virtual Machine, 58, 66
py2exe, 64
py2wasm, 45, 63
PyDoc, 113, 468, 912
 dostosowywanie pakietu, 474
 funkcja help, 468
 interfejs pakietu, 472
 raporty HTML, 472
 strona główna, 473
 użytkowanie, 475
 wyświetlanie dokumentacji, 474
Pydroid 3, 70
PyInstaller, 64
Pyjamas, 45
PyMongo, 46
Pyodide, 45
pyperf, 680
PyPy, 42, 46, 61, 62, 66
PyQt, 44
PyScript, 45
Pyston, 63
PyThran, 46, 63
pywin32, 45
PyYAML, 46

R

raporty HTML, 472
referencja, 901
 obiekt.atrybut, 828
referencje, 178–180, 246, 260, 308, 319, 348
 cykliczne, 182
 słabe, 192
 współdzielone, shared references, 183, 186,
 360, 364, 541, 543
reguła LEGB, 506, 904
reguły
 dotyczące zasięgów, 505
 tworzenia wcięć, 395
rekordy, 290, 839
rekurencja, 579, 607, 790, 910, 998
 a kolejki i stosy, 584
 a pętle, 582
 bezpośrednia, 582
 pośrednia, 582
 stosowanie, 580–583, 588
rekurencyjne importowanie, 799
REPL, read-eval-print loop, 68, 72, 74, 293
 uruchamianie, 782
reszta z dzielenia, 151
rozpakowywanie
 argumentów, 553
 rozszerzone sekwencji, 349, 354, 357–359
rozszerzanie
 metod, 856
 typów
 klasy podrzędne, 1009
 osadzanie, 1008
rozszerzenia
 numeryczne, 174
 opcjonalne, 774
rzadkie struktury danych, 273

S

samodzielne programy, 1285
SciPy, 46
sekwencje, 107, 127, 136, 194, 241, 248
 kodów znaków Unicode, 198
 mieszające, 638
 mutowalne, 246
 niemutowalne, 213
 pomijanie elementów, 425

- przypisanie, 349–351, 418
- przypisanie zaawansowane, 352
- rozpakowanie rozszerzone, 349, 354
 - gadżet, 358
 - pętla for, 359
 - przypadki brzegowe, 357
- skanowanie, 423
- ucieczki, escape sequence, 197, 199, 200, 201
- wirtualne, 436
- zmiana kolejności, 424
- serializacja obiektów, 301, 1196
- sesja interaktywna, 74, 93
- Shed Skin, 62, 66
- silnik wykonawczy, runtime engine, 58
- skanery plików, 433
- składanie
 - list, 255
 - słowników, 266
- składnia
 - await/async, 491
 - dopasowania argumentów, 547
- składowe, 827
- skrótowa analiza obiektów, 405
- skrypt, 40, 77, 86, 94, 95, 703
 - mierzący wydajność, 669
 - testu wydajnościowego, 684
- skrypty internetowe, 44
- sloty, 940, 1017
 - atrybut `__dict__`, 1019, 1024
 - deklaracja, 1018
 - generyczna obsługa, 1022
 - reguły, 1023
 - stosowanie, 1025
 - używanie, 1018
 - w klasach nadrzędnych, 1021, 1023
 - w klasach podrzędnych, 1023
- słabe odwołanie, 192
- słowniki, dictionary, 104, 120, 258, 282, 839
 - `__dict__`, 995
 - brakujące klucze, 274
 - generowanie wyników, 634
 - indeksowanie, 325
 - klucz-wartość, 259
 - kolejność wstawiania kluczy, 268
 - literały, 261
 - łączenie, 269
 - metody, 124, 126, 261, 263
 - modyfikacja w miejscu, 262
 - mutowalność, 121, 259, 262
 - operacje na odwzorowaniach, 121
 - operacje, 260
 - porównywanie, 312, 313
 - porównywanie rozmiarów, 280
 - przestrzeni nazw, 731, 837, 907, 1019
 - rozpakowywanie literałów, 265, 269
 - rzadkie struktury danych, 273
 - składane, 266, 454
 - sortowanie kluczy, 126, 269, 279
 - sprawdzanie kluczy, 124
 - symulowanie elastycznych list, 273
 - tworzenie, 264, 429
 - widoki, 276, 278
 - zagnieżdżanie, 122, 275
 - zastosowanie pętli for, 443
 - zestawy argumentów, 121
- słowo kluczowe, 368
 - await, 653
 - class, 134
 - False, 314, 315
 - metaclass, 1253
 - mode, 1169
 - True, 314, 315, 385, 400
 - yield, 775
- sortowanie, 565
 - kluczy słowników, 279
 - list, 252
 - pomiar prędkości, 681
- spacje, 394
- Sphinx, 476
- spójność, cohesion, 577, 578
- sprawdzanie
 - błędów, 1085, 1153
 - danych wejściowych, 342
 - identyczności obiektów, 311
 - równości wartości, 311
- sprzęganie, coupling, 577, 578
- SQLAlchemy, 66
- SQLObject, 46
- Stackless, 62
- standardowa dokumentacja, 476
- standardowy strumień wyjściowy, 373
- stos typu LIFO, 254, 585, 1135
- struktura programu, 703
- strukturalne dopasowywanie wzorców, 389, 391
- struktury danych, data structures, 102

- strumienie
 - polecień powłoki, 305
 - standardowe, 305, 373
- strumień wyjścia
 - przekierowanie, 376
 - automatyczne, 378
 - ręczne, 377
 - standardowy, 373
- surowe łańcuchy znaków, raw strings, 201
- SWIG, 45
- sztuczna inteligencja, 47

Ś

- ścieżka do pliku, 1189
- ścieżki wyszukiwania modułów, 711
 - elementy, 712
 - konfigurowanie, 714
 - sprawdzanie, 716
 - zmiana, 716
- śląd stosu, stack trace, 1078, 1146

T

- tablice, 117
 - asocjacyjne, 259
 - mieszające, 259
 - skoków, jump table, 388, 598
- techniki interfejsów klas, 896
- Terminal, 70
- Termux, 70
- testowanie, 47, 76, 1153
 - interaktywne, 797
 - kodu, 569, 1144
 - przeładowań rekurencyjnych, 787
 - wariantów przeładowania, 791
- testy
 - jednostkowe, 776
 - logiczne, 314, 386, 399, 956
 - prawdziwości, 399
 - przynależności, 934
 - wydajnościowe, 682
- tkinter, 953
- Toga, 44
- transpilator Pyjamas, 45
- TurboGears, 45

- tworzenie
 - drzewa
 - atrybutów, 892
 - klas, 814
 - funkcji, 489
 - instancji, 845
 - klas, 845
 - podrzędnych, 855
 - wyjątków, 1117
 - kodu metod, 851
 - konstruktorów, 845
 - metaklas, 1224, 1226
 - modułów, 721
 - obiektów bytes, 1183
 - pakietów, 744
 - słowników, 264, 429
 - wycinków, 960
 - zbioru, 167
- typ
 - bool, 139, 315
 - bytearray, 1168, 1209
 - bytes, 1168, 1181, 1209
 - class, 1012
 - Decimal, 162
 - Fraction, 164
 - str, 1167, 1181, 1209
- typy, 1012
 - danych, 103
 - dynamiczne, 49, 105, 177, 188
 - iterowalne, 443
 - jednostek programu, 104
 - konwersje, 1171
 - liczbowe, 138
 - mutowalne, 242
 - niemutowalne, 242, 321, 326
 - podstawowe obiektów, 104, 131, 136, 317
 - krotki, 127, 285
 - liczby, 105, 138
 - listy, 116, 245
 - łańcuchy znaków, 107, 193
 - pliki, 128, 291
 - słowniki, 120, 258
 - zbiory, 131
 - powiązane z implementacją, 104
 - sekwencyjne, 116
 - silne, 105
 - wbudowane, 50, 316
 - rozszerzanie, 1008

U

- udostępnianie kodu, 1155
- ukrywanie danych, 754, 771
- Unicode, 114, 130, 193, 198, 298, 1161–3, 1166,
Patrz także ASCII
 - dekodowanie, 1173–78
 - domyślne kodowanie, 1200
 - kodowanie znaków, 1162, 1164, 1173–78
 - łańcuchy szerokoznakowe, 1163
 - normalizacja, 1205
 - obsługa BOM, 1201
 - pliki tekstowe, 1191
 - punkty kodowe, 1163, 1167
 - tekst, 1167, 1210
 - typ str, 1167
- uruchamianie programów, 54, 67, 78
 - ikona programu, 81
 - import modułów, 86
 - nazwa pliku, 81
 - opcje, 91
 - przez inny kod, 91
 - środowisko IDLE, 82
 - wiersz poleceń, 79
- UTF-8, 1165, 1171, 1191, 1198, 1200
- UTF-16, 1166
- UTF-32, 1166

W

- wartości
 - domyślne, 690
 - logiczne, 132, 173
- wartość `__main__`, 775
- wcięcia, 334, 339, 395, 396
- WebAssembly, 44, 45, 84
- wiązania POSIX, 44
- widok, 271
 - items, 278
 - keys, 278
- widoki słowników, 276, 278
- widzety GUI, 318
- wielowątkowość, 516
- wiersz
 - kontynuacji, 397, 398
 - poleceń, command-line, 44, 70
 - interaktywny, 68
 - sposoby użycia, 79
 - uruchamianie programu, 78

- wirtualne
 - obiekty iterowalne, 437
 - sekwencje, 436
- właściwości klas, 1027
- wskaźnik, 180, 541, 543
- współprogramy, coroutines, 491, 661
- wxPython, 44
- wycinki, slice, 108, 206, 244, 324, 425, 919, 960
 - rozszerzone, 209
- wydajność, 38, 39, 59, 664, 669
- wyjątek, exception, 343, 1075
 - AssertionError, 1115
 - AttributeError, 938
 - EOFError, 1141
 - StopIteration, 437, 440, 460, 620, 775
- wyjątki
 - domyślne wyświetlanie, 1124
 - domyślny program obsługi, 1077
 - grupy, 1130
 - hierarchie, 1119
 - kategorie, 1123
 - klasy wbudowane, 1122
 - łańcuchy, 1104
 - przechwytywanie, 1078
 - zbyt mało, 1150
 - zbyt wiele, 1148
 - przekazywanie, 1103
 - role, 1076
 - stan, 1124
 - sygnalizowanie wyników, 1141, 1142
 - tworzenie klas, 1117
 - udostępnianie metod, 1128
 - udostępnianie szczegółów, 1127
 - używanie, 1075
 - wbudowane, 1093
 - własne dane, 1127
 - własne sposoby wyświetlania, 1126
 - zagnieżdżanie, 1135
 - zastosowanie, 1140
 - zdefiniowane przez użytkownika, 1081
 - zgłaszanie, 1080, 1101
- wykonywanie
 - programu, 55, 59
 - programu interaktywne, 73
- wyniki
 - działania skryptu testującego, 685
 - pomiarów czasu, 671

- wyrażenia, expression, 73, 101, 141, 329
 - await, 652
 - formatujące
 - łańcuchy znaków, 224
 - oparte na słowniku, 226
 - zaawansowane, 225
 - generatorów, 454, 457, 459, 619, 625, 630, 640, 926
 - a filtry, 629
 - a funkcja map, 628
 - zalety, 627
 - indeksujące, 107
 - jako instrukcje, 371
 - lambda, 388, 490, 494, 495, 509, 596, 607
 - forma, 596
 - pętle, 600
 - tworzenie tablic skoków, 598
 - używanie, 597
 - wywołania zwrotne, 608
 - zagnieżdżanie, 601
 - list składanych, 119
 - podstawowe, 146
 - przypisania nazwanego, 349, 350, 364
 - self.atrybut, 828
 - trójargumentowe if/else, 401, 402
 - yield, 142, 623
 - w nawiasach, 144
 - złożone, 124
 - z literałem, 104, 246, 281
 - z wycinkiem, 210, 426
- wyszukiwanie
 - atrybutów dziedziczonych, 810
 - dziedziczenia, 829, 837, 901
 - dziedziczenia MRO, 984, 986
 - komponentów, 821
- wyświetlanie
 - atrybutów, 1000
 - błędów i śladów stosu, 1146
 - obiektów, 943, 944, 960
 - wyjątków, 1124, 1126
- wywołania, 951
 - bezpośrednie, 783
 - funkcji, 73, 374
 - metod klasy, 214, 813, 892
 - obiektu klasy, 825
 - zwrotne, 953, 980
- wzorce, 391
 - alternatywy, 391
 - atrybutów i instancji, 392
 - literałowe, 391
 - przechwytywania, 391
 - uniwersalne, 391
 - z przypisaniem, 391
- wzorzec
 - diamentowy, 986
 - projektowy, design pattern, 521, 820
 - delegacja, 971, 973
 - dziedziczenie, 964
 - dziedziczenie wielokrotne, 984
 - fabryka, 981
 - kompozycja, 966

Z

- zaciemnianie
 - kodu, 600, 617
 - znaczenia zmiennych, 796
- zadania
 - równoległe, 653
 - sekwencyjne, 652
 - współbieżne, 656, 658
- zagnieżdżanie, 326, 346
 - instrukcji try/except, 1136
 - instrukcji try/finally, 1136
 - iteratorów, 445
 - klas, 904
 - obiektów, 118, 122, 307
 - przebiegu sterowania, 1137
 - przestrzeni nazw, 734
 - składniowe, 1138
 - słowników, 275
 - wyrażeń lambda, 601
 - zasięgów, 523
- zalety Pythona, 1257
- zapętlenie rekurencyjne, 996
- zarządzanie
 - instalacją, 1155
 - kontekstem, 1110
- zasięgi, scope, 503, 532
 - dowolne zagnieżdżanie, 523
 - dynamiczne, 734
 - globalne, 505, 509
 - LEGB, 508
 - leksykalne, 504, 734

- lokalne, 505, 510
 - funkcji, 914
 - klasy, 914
- wbudowane, 511
- zagnieżdżone, 519
- zbieranie argumentów, 552
- zbiory, set, 104, 131, 139, 166, 278, 283, 306
 - izolowanie różnic, 171
 - metody, 168
 - mutowalne, 132
 - niemutowalne, 131, 170
 - odfiltrowywanie duplikatów, 171
 - porównywanie, 312
 - przetwarzanie zestawów danych, 172
 - puste, 167
 - składane, 170, 454
 - śledzenie miejsc, 172
 - testy równości, 171
 - tworzenie, 167
 - zamrożone, 169
- zintegrowane środowisko programistyczne, IDE, 67, 82, 1154
- zmienna
 - `__name__`, 775, 776, 801
 - `__slots__`, 1018
 - środowiskowa
 - PATH, 80
 - PYTHONPATH, 718
 - PYTHONSTARTUP, 782
- zmiennie, 146, 178–180
 - globalne, 504, 515, 531
 - deklaracja, 518
 - metody dostępu, 518
 - minimalizowanie stosowania, 515
 - modyfikowalne, 528
 - współdzielone, 528
- lokalne, 500, 504
- miejsce przypisania, 902
- nonlocal, 526, 527
 - LEGB, 527
 - modyfikowalne, 527
 - na wywołanie, 527
- prywatne, 941
- składane, 508, 636
- wybór nazw, 367
- wyjątków, 508
- znacznik kolejności bajtów, BOM, 1201
- znaki
 - #, 72, 78, 146, 394, 462
 - \$, 71, 240
 - %, 222
 - ** , 105, 266, 553, 554
 - @, 143, 1041
 - +, 105, 109, 248, 257
 - białe, 594
 - cudzysłowów, 236
 - dekodowanie, 1164
 - dwukropka, 332, 347
 - emotikony, 1163
 - gwiazdki, *, 75, 105, 359, 553–555
 - kodowanie, 1162, 1164
 - ASCII, 1162
 - Unicode, 1163, 1173
 - kontynuacji, 397
 - lewego ukośnika, \, 197, 397, 398
 - nowego wiersza, \n, 114, 197, 399, 1189
 - podkreślenia, 112, 158, 771, 800
 - podwójnego podkreślenia, 592, 832
 - prawych ukośników, 202
 - specjalne, 197
 - średnika, 333, 397
 - tabulatora, \t, 197
 - trzech apostrofów, 203
 - trzech cudzysłowów, 204
 - trzech kropek, 410
 - ucieczki, 113
 - zachęty
 - ..., 71, 203, 385
 - >>>, 70, 73
 - >>>>, 72
 - znikształcanie nazw zmiennych, 974
 - ZODB, 46
 - Zope, 45
 - zorientowanie
 - obiektowe, 48, 135
 - obiektowo-skryptowe, 40
 - związek
 - typu „jest”, 964
 - typu „ma”, 966

Ż

źródła dokumentacji, 461

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

Nic dziwnego, że programiści kochają Pythona: jest wszechstronny, czytelny, darmowy i działa na każdej platformie. Można się go stosunkowo szybko nauczyć — ale jest jeden warunek: aby w pełni wykorzystać jego możliwości, trzeba zdobyć solidne podstawy, zrozumieć kilka trudniejszych koncepcji i... dużo ćwiczyć, pisząc własny kod.

Ta książka stanowi kompleksowe i obszerne wprowadzenie do języka Python. Pomoże Ci opanować jego podstawy i przygotuje do praktycznego zastosowania nabytej wiedzy. To wydanie zostało zaktualizowane i rozszerzone, aby odzwierciedlić zmiany zachodzące w świecie Pythona. Pominęto omówienie nieaktualnej wersji 2.X, opisano nowe narzędzia, dodane do Pythona w wersji 3.12, a także innych jego edycji, które są dziś szeroko używane. Naukę ułatwią Ci liczne quizy, ćwiczenia, pomocne ilustracje i przykładowe fragmenty kodu. To idealne kompendium dla każdego, kto chce szybko zacząć programować w Pythonie i tworzyć wydajny kod wysokiej jakości.

Zaufaj Pythonowi i odkryj świat programowania!

W książce:

- ogólny model składni Pythona
- wbudowane typy obiektów i ich przetwarzanie
- stosowanie funkcji i organizowanie kodu w modułach i pakietach
- obsługa wyjątków i inne narzędzia programistyczne
- zaawansowane narzędzia Pythona, między innymi dekoratory, deskryptory i metaklasy
- kod idiomatyczny, który działa na różnych platformach

Mark Lutz od niemal trzydziestu lat zajmuje się nauczaniem Pythona. Jest autorem niezwykle popularnych i wielokrotnie wznawianych książek poświęconych temu językowi programowania, przeprowadził też setki sesji treningowych w tym zakresie. Obecnie tworzy w Pythonie aplikacje, które działają zarówno na komputerach osobistych, jak i telefonach.


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

KOD KORZYŚCI
Siegnij po więcej! ▶



ISBN 978-83-289-2942-5



Cena: 199,00 zł