

SOFTWARE DEVELOPMENT SERIES  
SCOTT MEYERS, KONSULTANT REDAKCYJNY



# *Efektywny* PYTHON

*125 sposobów na lepszy kod*

WYDANIE III



Brett Slatkin

Helion

Tytuł oryginału: Effective Python: 125 Specific Ways to Write Better Python, 3rd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-2695-0

Authorized translation from the English language edition, entitled EFFECTIVE PYTHON: 125 SPECIFIC WAYS TO WRITE BETTER PYTHON, 3rd Edition 9780138172183 by Brett Slatkin published by Pearson Education, Inc, publishing as Addison-Wesley Professional.

Copyright © 2025 Pearson Education, Inc.

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 Pearson Education, Inc.

Polish language edition published by Helion S.A., Copyright © 2025

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.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: [helion.pl](http://helion.pl) (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[helion.pl/user/opinie/efpyt3](http://helion.pl/user/opinie/efpyt3)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

# Spis treści

<b>Wprowadzenie</b> .....	<b>11</b>
<b>Podziękowania</b> .....	<b>17</b>
<b>O autorze</b> .....	<b>19</b>
<b>Rozdział 1. Programowanie zgodne z duchem Pythona</b> .....	<b>21</b>
Sposób 1. Ustalenie używanej wersji Pythona .....	21
Sposób 2. Stosuj styl PEP 8 .....	23
Sposób 3. Nigdy nie oczekuj, że Python wykryje błędy podczas kompilacji .....	26
Sposób 4. Decyduj się na funkcje pomocnicze zamiast na skomplikowane wyrażenia ....	28
Sposób 5. Zamiast indeksowania wybieraj rozpakowanie wielu operacji przypisania ....	31
Sposób 6. Krotkę jednoelementową zawsze umieszczaj w nawiasie .....	35
Sposób 7. W prostej logice osadzonej używaj wyrażen warunkowych .....	38
Sposób 8. Unikaj powtórzeń w wyrażeniach przypisania .....	42
Sposób 9. Stosuj polecenie match w celu destrukuryzacji kontroli przepływu, ale unikaj go, gdy wystarczające będzie polecenie if .....	48
<b>Rozdział 2. Ciągi tekstowe i wycinki</b> .....	<b>56</b>
Sposób 10. Różnice między typami bytes i str .....	56
Sposób 11. Wybieraj interpolowane ciągi tekstowe f zamiast ciągów tekstowych formatowania w stylu C i funkcji str.format() .....	61
Sposób 12. Różnice między wywołaniami repr i str podczas wyświetlania obiektów .....	72
Sposób 13. Wybieraj jawną konkatenację ciągu tekstowego zamiast niejawnej, zwłaszcza w przypadku listy .....	76
Sposób 14. Umiejętnie podziel sekwencje .....	80
Sposób 15. Unikaj użycia indeksów początek, koniec i wartości kroku w pojedynczej operacji podziału .....	82
Sposób 16. Wybieraj rozpakowanie typu catch-all zamiast tworzenia wycinków .....	85

<b>Rozdział 3. Pętle i iteratory .....</b>	<b>89</b>
Sposób 17. Preferuj użycie funkcji enumerate() zamiast range() .....	89
Sposób 18. Używaj funkcji zip() do równoczesnego przetwarzania iteratorów .....	91
Sposób 19. Unikaj bloków else po pętlach for i while .....	94
Sposób 20. Nigdy nie używaj zmiennych pętli for po zakończeniu jej działania .....	96
Sposób 21. Podczas iteracji przez argumenty zachowuj postawę defensywną .....	98
Sposób 22. Nigdy nie modyfikuj kontenerów podczas iteracji przez nie, lecz skorzystaj z kopii lub pamięci podręcznej .....	103
Sposób 23. Przekazuj iteratory do wywołań any() i all() w celu skrócenia logiki .....	108
Sposób 24. Rozważ stosowanie modułu itertools w pracy z iteratorami i generatorami .....	112
<b>Rozdział 4. Słowniki .....</b>	<b>119</b>
Sposób 25. Zachowaj ostrożność, gdy polegasz na kolejności wstawiania elementów do obiektu typu dict .....	119
Sposób 26. Podczas obsługi brakujących kluczy słownika wybieraj funkcję get() zamiast operatora in i wyjątku KeyError .....	126
Sposób 27. Podczas obsługi brakujących elementów w wewnętrznym stanie wybieraj typ defaultdict zamiast metody setdefault() .....	131
Sposób 28. Wykorzystaj metodę __missing__() do tworzenia wartości domyślnych w zależności od klucza .....	133
Sposób 29. Twórz klasy, zamiast zagnieżdżać wiele poziomów słowników, list i krotek .....	136
<b>Rozdział 5. Funkcje .....</b>	<b>142</b>
Sposób 30. Ustal, kiedy argumenty funkcji mogą być zmienione .....	142
Sposób 31. Zwróć obiekt, gdy funkcja ma rozpakować więcej niż trzy zmienne .....	145
Sposób 32. Preferuj wyjątki zamiast zwrotu wartości None .....	148
Sposób 33. Zobacz, jak domknięcia współdziałają z zakresem zmiennej .....	151
Sposób 34. Zmniejszenie wizualnego zagmatwania za pomocą zmiennej liczby argumentów pozycyjnych .....	155
Sposób 35. Zdefiniowanie zachowania opcjonalnego za pomocą argumentów w postaci słów kluczowych .....	158
Sposób 36. Użycie None i docstring w celu dynamicznego określenia argumentów domyślnych .....	162
Sposób 37. Wymuszaj czytelność kodu za pomocą argumentów w postaci słów kluczowych i argumentów jedynie pozycyjnych .....	165
Sposób 38. Dekoratory funkcji definiuj za pomocą functools.wraps .....	170
Sposób 39. Podczas łączenia funkcji preferuj functools.partial zamiast wyrażeń lambda .....	173

<b>Rozdział 6. Konstrukcje składane i generatory .....</b>	<b>176</b>
Sposób 40. Używaj list składanych zamiast funkcji map() i filter() .....	176
Sposób 41. Unikaj więcej niż dwóch wyrażeń na liście składanej .....	178
Sposób 42. Stosuj wyrażenia przypisania, aby unikać powielania zadań w konstrukcjach składanych .....	180
Sposób 43. Rozważ użycie generatorów, zamiast zwracać listy .....	183
Sposób 44. Rozważ użycie generatora wyrażeń dla dużych list składanych .....	186
Sposób 45. Twórz wiele generatorów za pomocą wyrażenia yield from .....	188
Sposób 46. Iteratory przekazuj generatorom jako argumenty zamiast za pomocą metody send() .....	190
Sposób 47. Przejścia między stanami obsługi za pomocą klasy, zamiast używać metody throw() .....	195
<b>Rozdział 7. Klasy i interfejsy .....</b>	<b>200</b>
Sposób 48. Dla prostych interfejsów akceptuj funkcje zamiast klas .....	200
Sposób 49. Wybieraj zorientowany obiektowo polimorfizm zamiast funkcji z wywołaniami isinstance() .....	204
Sposób 50. W stylu programowania funkcyjnego używaj functools.singledispatch zamiast polimorfizmu zorientowanego obiektowo .....	208
Sposób 51. Używaj typu dataclasses zamiast lekkich klas .....	215
Sposób 52. Użycie polimorfizmu @classmethod w celu ogólnego tworzenia obiektów .....	227
Sposób 53. Inicjalizacja klasy nadrzędnej za pomocą wywołania super() .....	231
Sposób 54. Rozważ łączenie funkcjonalności za pomocą klas domieszek .....	235
Sposób 55. Preferuj atrybuty publiczne zamiast prywatnych .....	240
Sposób 56. Wybieraj moduł dataclasses, zamiast tworzyć niemodyfikowalne obiekty .....	245
Sposób 57. Stosuj dziedziczenie po collections.abc w kontenerach typów niestandardowych .....	253
<b>Rozdział 8. Metaklasy i atrybuty .....</b>	<b>258</b>
Sposób 58. Używaj zwykłych atrybutów zamiast metod typu getter i setter .....	258
Sposób 59. Rozważ użycie @property zamiast refaktoryzacji atrybutów .....	262
Sposób 60. Stosuj deskryptory, aby wielokrotnie wykorzystywać metody udekorowane przez @property .....	266
Sposób 61. Używaj metod __getattr__(), __getattribute__() i __setattr__() dla opóźnionych atrybutów .....	271
Sposób 62. Sprawdzaj podklasy za pomocą __init_subclass__ .....	277
Sposób 63. Rejestruj istniejące klasy za pomocą __init_subclass__() .....	283
Sposób 64. Adnotacje atrybutów klas dodawaj za pomocą metody __set_name__() ...	288
Sposób 65. Przemyśl kolejność definicji klasy, aby zdefiniować powiązania między atrybutami .....	292
Sposób 66. Dla złożonych rozszerzeń klas wybieraj dekoratory klas zamiast metaklas .....	298

<b>Rozdział 9. Współbieżność i równoległość .....</b>	<b>304</b>
Sposób 67. Używaj modułu subprocess do zarządzania procesami potomnymi .....	305
Sposób 68. Użycie wątków dla operacji blokujących wejście-wyjście, unikanie równoległości .....	309
Sposób 69. Używaj klasy Lock, aby unikać stanu wyścigu w wątkach .....	314
Sposób 70. Używaj klasy Queue do koordynacji pracy między wątkami .....	317
Sposób 71. Naucz się rozpoznawać, kiedy współbieżność jest niezbędna .....	327
Sposób 72. Unikaj tworzenia nowych egzemplarzy Thread na żądanie fan-out .....	331
Sposób 73. Pamiętaj, że stosowanie Queue do obsługi współbieżności wymaga refaktoringu .....	335
Sposób 74. Rozważ użycie klasy ThreadPoolExecutor, gdy wątki są potrzebne do zapewnienia współbieżności .....	341
Sposób 75. Zapewnij wysoką współbieżność operacji wejścia-wyjścia dzięki użyciu współprogramów .....	344
Sposób 76. Naucz się przekazywać do asyncio wątkowane operacje wejścia-wyjścia ....	348
Sposób 77. Połączenie wątków i współprogramów w celu ułatwienia konwersji na wersję stosującą asyncio .....	359
Sposób 78. Maksymalizuj responsywność przez unikanie blokującej pętli zdarzeń asyncio .....	365
Sposób 79. Rozważ użycie concurrent.futures(), aby otrzymać prawdziwą równoległość .....	369
<b>Rozdział 10. Niezawodność .....</b>	<b>373</b>
Sposób 80. Wykorzystaj zalety wszystkich bloków w konstrukcji try-except-else-finally .....	373
Sposób 81. Używaj polecenia assert dla wewnętrznych założeń i raise w przypadku niespełnionych oczekiwań .....	378
Sposób 82. Rozważ użycie poleceń contextlib i with w celu uzyskania wielokrotnego użycia konstrukcji try-finally .....	381
Sposób 83. Zawsze staraj się maksymalnie skracać blok try .....	385
Sposób 84. Uważaj na znikające zmienne wyjątku .....	387
Sposób 85. Uważaj podczas przechwytywania klasy Exception .....	388
Sposób 86. Poznaj różnicę między klasami Exception i BaseException .....	391
Sposób 87. Używaj modułu traceback w celu dostarczania dokładniejszych informacji o wyjątkach .....	396
Sposób 88. Rozważ jawne łączenie wyjątków, aby otrzymać przejrzyste stosy wywołań .....	400
Sposób 89. Zawsze przekazuj zasoby do generatorów i nakazuj komponentom wywołującym przeprowadzanie operacji porządkowych .....	407
Sposób 90. Nigdy nie przypisuj wartości False zmiennej __debug__ .....	412
Sposób 91. Unikaj wywołań exec() i eval(), o ile nie stworzysz narzędzia programistycznego .....	414

<b>Rozdział 11. Wydajność .....</b>	<b>417</b>
Sposób 92. Przed optymalizacją przeprowadzaj profilowanie .....	418
Sposób 93. Wydajność działania kodu o znaczeniu krytycznym optymalizuj za pomocą testów wydajności modułu <code>timeit</code> .....	423
Sposób 94. Dostrzegaj, kiedy i jak zastąpić Pythona innym językiem programowania .....	427
Sposób 95. Rozważ użycie modułu <code>ctypes</code> w celu sprawnej integracji z bibliotekami natywnymi .....	431
Sposób 96. Rozważ użycie modułów rozszerzeń, aby zmaksymalizować wydajność działania i ergonomię .....	436
Sposób 97. Aby skrócić czas uruchamiania programu, korzystaj ze wstępnie skompilowanego kodu bajtowego i buforowania systemu plików .....	442
Sposób 98. Stosuj wczytywanie modułów z opóźnieniem i importowanie dynamiczne, aby skrócić czas uruchamiania programu .....	445
Sposób 99. Rozważ użycie typów <code>memoryview</code> i <code>bytearray</code> , gdy podczas pracy z typem <code>bytes</code> stosujesz tzw. kopiowanie zero .....	451
<b>Rozdział 12. Algorytmy i struktury danych .....</b>	<b>458</b>
Sposób 100. Używaj parametru <code>key</code> podczas sortowania według skomplikowanych kryteriów .....	458
Sposób 101. Poznaj różnice między metodami <code>sort()</code> i <code>sorted()</code> .....	464
Sposób 102. Podczas wyszukiwania danych w sortowanych sekwencjach stosuj moduł <code>bisect</code> .....	465
Sposób 103. Wybieraj typ <code>deque</code> podczas tworzenia kolejek typu producent – konsument .....	468
Sposób 104. Przekonaj się, jak używać <code>heapq</code> w kolejce priorytetowej .....	473
Sposób 105. Podczas obsługi czasu lokalnego używaj modułu <code>datetime</code> zamiast <code>time</code> .....	481
Sposób 106. Gdy ważna jest precyzja, używaj modułu <code>decimal</code> .....	485
Sposób 107. Niezawodne użycie <code>pickle</code> wraz z <code>copyreg</code> .....	488
<b>Rozdział 13. Testowanie i debugowanie .....</b>	<b>495</b>
Sposób 108. W podklasach klasy <code>TestCase</code> sprawdzaj powiązane ze sobą zachowanie .....	496
Sposób 109. Wybieraj testy integracyjne zamiast testów jednostkowych .....	502
Sposób 110. Izoluj testy od siebie za pomocą metod <code>setUp()</code> , <code>tearDown()</code> , <code>setUpModule()</code> i <code>tearDownModule()</code> .....	507
Sposób 111. Podczas testowania kodu zawierającego skomplikowane zależności korzystaj z imitacji .....	509
Sposób 112. Hermetyzuj zależności, aby ułatwić tworzenie imitacji i testowanie .....	517



Sposób 113. Używaj metody <code>assertAlmostEqual()</code> do kontrolowania precyzji w testach liczb zmiennoprzecinkowych .....	520
Sposób 114. Rozważ interaktywne usuwanie błędów za pomocą <code>pdb</code> .....	523
Sposób 115. Stosuj moduł <code>tracemalloc</code> , aby poznać sposób użycia pamięci i wykryć jej wycieki .....	527
<b>Rozdział 14. Współpraca .....</b>	<b>531</b>
Sposób 116. Kiedy szukać modułów opracowanych przez społeczność? .....	531
Sposób 117. Używaj środowisk wirtualnych dla odizolowanych i powtarzalnych zależności .....	532
Sposób 118. Dla każdej funkcji, klasy i modułu utwórz <code>docstring</code> .....	537
Sposób 119. Używaj pakietów do organizacji modułów i dostarczania stabilnych API .....	542
Sposób 120. Rozważ użycie kodu o zasięgu modułu w celu konfiguracji środowiska wdrożenia .....	547
Sposób 121. Zdefiniuj główny wyjątek <code>Exception</code> w celu odizolowania komponentu wywołującego od API .....	549
Sposób 122. Zobacz, jak przerwać krąg zależności .....	553
Sposób 123. Rozważ użycie modułu <code>warnings</code> podczas refaktoryzacji i migracji kodu .....	558
Sposób 124. Rozważ stosowanie analizy statycznej za pomocą modułu <code>typing</code> w celu usuwania błędów .....	565
Sposób 125. Podczas tworzenia paczek programów w Pythonie wybieraj projekty otwartoźródłowe zamiast modułów <code>zipimport</code> i <code>zipapp</code> .....	571





## Sposób 3. Nigdy nie oczekuj, że Python wykryje błędy podczas kompilacji

Podczas wczytywania programu i przygotowywania go do uruchomienia kod źródłowy programu zostaje przekształcony na postać drzewa składniowego i sprawdzony pod kątem oczywistych błędów strukturalnych. Na przykład kiepsko przygotowana konstrukcja `if` spowoduje zgłoszenie wyjątku `SyntaxError`, który wskazuje na istnienie błędu w kodzie.

```
if True # Niepoprawna składnia
    print("witaj")
```

```
>>>
Traceback ...
SyntaxError: expected ':'
```

Błędy w literałach również zostaną wychwycone na wczesnym etapie i doprowadzą do zgłoszenia odpowiednich wyjątków.

```
1.3j5 # Niepoprawna liczba
```

```
>>>
Traceback ...
SyntaxError: invalid imaginary literal
```

Niestety to wszystko w zakresie wykrywania błędów, na co można liczyć od Pythona przed uruchomieniem programu. Wszelkie nieprawidłowości wykraczające poza błędy tokenizacji i przetwarzania nie będą zgłaszane.

Nawet w przypadku prostych funkcji zawierających oczywiste błędy nie należy oczekiwać ich zgłoszenia przed uruchomieniem programu. To wiąże się z wysoce dynamiczną naturą Pythona. Na przykład spójrz na definicję funkcji, w której zmiennej `my_var` nie została przypisana wartość przed przekazaniem tej zmiennej do funkcji `print()`.

```
def bad_reference():
    print(my_var)
    my_var = 123
```

Jednak mimo to zgłoszenie wyjątku nie nastąpi aż do chwili wywołania funkcji.

```
bad_reference()
```

```
>>>
```

```
Traceback ...
```

```
UnboundLocalError: cannot access local variable 'my_var' where it is not associated with a value
```

Tego rodzaju błąd nie jest uznawany za *statyczny*, ponieważ w programach Pythona można dynamicznie przypisywać wartości zmiennym lokalnym i globalnym. W kolejnym fragmencie kodu zamieściłem definicję funkcji, która jest poprawna i niezależna od argumentu danych wejściowych.

```
def sometimes_ok(x):
    if x:
        my_var = 123
    print(my_var)
```

Przedstawione tutaj wywołanie funkcji działa bezbłędnie.

```
sometimes_ok(True)
```

```
>>>
```

```
123
```

Natomiast kolejne prowadzi do zgłoszenia wyjątku.

```
sometimes_ok(False)
```

```
>>>
```

```
Traceback ...
```

```
UnboundLocalError: cannot access local variable 'my_var' where it is not associated with a value
```

Python nie wychwyci z wyprzedzeniem także błędów matematycznych. Dlatego też interpreter nie będzie wiedział o istnieniu błędu w tej funkcji, dopóki nie spróbuje jej wykonać.

```
def bad_math():
    return 1 / 0
```

Sposób działania operatora dzielenia zależy od wartości operandów, więc sprawdzenie pod kątem błędów zostaje odłożone aż do chwili wykonywania tego kodu.

```
bad_math():  
  
>>>  
Traceback ...  
ZeroDivisionError: division by zero
```

Python nie będzie również statycznie wykrywał innych problemów, takich jak niezdefiniowana metoda, zbyt duża lub zbyt mała liczba przekazanych argumentów, niepoprawny typ wartości zwrotnej i wiele innych, wydawałoby się oczywistych, problemów. Na szczęście społeczność opracowała różne narzędzia, które mogą pomóc w wychwytywaniu przynajmniej części tego rodzaju błędów. Przykładem takiego narzędzia jest linter `flake8` (<https://github.com/PyCQA/flake8>). Ponadto mamy narzędzia do sprawdzania typów, które współdziałają z wbudowanym modulem `typing` (więcej informacji na ten temat przedstawię w sposobie 124.).

Ostatecznie podczas tworzenia kodu zgodnego z duchem Pythona większość błędów napotkasz po uruchomieniu programu. Twórcy języka przywiązują większą wagę do elastyczności Pythona w trakcie działania programu niż do wykrywania błędów podczas kompilacji. Z tego powodu po uruchomieniu programu trzeba koniecznie sprawdzać poprawność przyjętych założeń (zobacz sposób 81.) oraz zweryfikować poprawność kodu źródłowego za pomocą testów zautomatyzowanych (zobacz sposób 109.).

## Do zapamiętania

- ◆ Niemal wszystkie operacje sprawdzania pod kątem błędów Python odracza aż do chwili uruchomienia programu. To dotyczy również problemów, których istnienie wydaje się oczywiste już na etapie tworzenia programu.
- ◆ Opracowane przez społeczność projekty, takie jak lintery i narzędzia do statycznej analizy kodu źródłowego, mogą pomóc w wychwyceniu niektórych z najczęściej popełnianych błędów jeszcze przed uruchomieniem programu.

# 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** 

# Programujesz w Pythonie?

## Poznaj 125 sposobów na lepszy kod!

Python cieszy się dużym uznaniem. Jest wszechstronny i efektywny, a przy tym konsekwentnie rozwijany. Język ten ma również wiele trudniejszych do uchwycenia zalet. Aby uzyskać imponujące efekty wydajności kodu, jego wieloplatformowość i bezpieczeństwo, a przy tym w pełni korzystać z możliwości Pythona, potrzebujesz czegoś więcej niż tylko znajomość jego składni.

To trzecie, zaktualizowane i uzupełnione wydanie lubianego podręcznika programowania w Pythonie. Zawiera dodatkowe rozdziały dotyczące tak ważnych zagadnień jak algorytmy i struktury danych. Zawarty w książce materiał, w tym słowniki, ułatwi Ci tworzenie solidnego, wydajnego kodu. Znajdziesz tu jasne, zwięzłe i praktyczne rady przeznaczone dla programistów na różnym poziomie zaawansowania. Niezależnie od tego, czy tworzysz aplikacje internetowe, analizujesz dane, czy trenujesz modele sztucznej inteligencji, dzięki temu podręcznikowi zdobędziesz cenne umiejętności pracy z Pythonem!

### W książce:

- 125 gotowych do użycia sposobów na lepszy kod
- uaktualnione dobre praktyki, uwzględniające nowości w Pythonie do wersji 3.13
- dodatkowe rozdziały poświęcone tworzeniu niezawodnych, wysoko wydajnych programów
- tworzenie modułów rozszerzeń w języku C
- współpraca z natywnymi bibliotekami współdzielonymi
- praktyczne przykłady kodu odzwierciedlające najlepsze praktyki

**Brett Slatkin** jest głównym inżynierem oprogramowania w firmie Google. Pracował nad projektem Google Surveys, nad protokołem PubSubHubbub, a wcześniej zajmował się pierwszym w Google produktem przetwarzania w chmurze — App Engine. Ukończył Uniwersytet Columbia w Nowym Jorku, programuje w Pythonie od blisko 20 lat.

	<b>KOD KORZYŚCI</b> Sięgnij po więcej! ▶
helion.pl	
HELION S.A. ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	ISBN 978-83-289-2695-0
9 788328 926950	
Cena: 139,00 zł	

**Pearson**  
Addison-Wesley