

*Sprawdzone przepisy
dla programistów Pythona!*

Wydanie III



Python Receptury



O'REILLY®

David Beazley, Brian K. Jones

Tytuł oryginału: Python Cookbook, 3rd Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-246-8180-8

© 2014 Helion S.A.

Authorized Polish translation of the English edition of Python Cookbook, 3rd Edition, ISBN 9781449340377 © 2013 David Beazley, Brian Jones.

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/pytre3>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/pytre3.zip>

Printed in Poland.

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

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

Spis treści

Przedmowa	11
1. Algorytmy i struktury danych	15
1.1. Wypakowywanie sekwencji do odrębnych zmiennych	15
1.2. Wypakowywanie elementów z obiektów iterowalnych o dowolnej długości	16
1.3. Zachowywanie ostatnich N elementów	19
1.4. Wyszukiwanie N największych lub najmniejszych elementów	20
1.5. Tworzenie kolejki priorytetowej	22
1.6. Odwzorowywanie kluczy na różne wartości ze słownika	24
1.7. Określanie uporządkowania w słownikach	25
1.8. Obliczenia na danych ze słowników	26
1.9. Wyszukiwanie identycznych danych w dwóch słownikach	28
1.10. Usuwanie powtórzeń z sekwencji przy zachowaniu kolejności elementów	29
1.11. Nazywanie wycinków	30
1.12. Określanie najczęściej występujących w sekwencji elementów	31
1.13. Sortowanie list słowników według wspólnych kluczy	33
1.14. Sortowanie obiektów bez wbudowanej obsługi porównań	34
1.15. Grupowanie rekordów na podstawie wartości pola	35
1.16. Filtrowanie elementów sekwencji	37
1.17. Pobieranie podzbioru słownika	39
1.18. Odwzorowywanie nazw na elementy sekwencji	40
1.19. Jednoczesne przekształcanie i redukcja danych	42
1.20. Łączenie wielu odwzorowań w jedno	43
2. Łańcuchy znaków i tekst	47
2.1. Podział łańcuchów znaków po wykryciu dowolnego z różnych ograniczników	47
2.2. Dopasowywanie tekstu do początkowej lub końcowej części łańcucha znaków	48
2.3. Dopasowywanie łańcuchów znaków za pomocą symboli wieloznacznych powłoki	50
2.4. Dopasowywanie i wyszukiwanie wzorców tekstowych	51

2.5. Wyszukiwanie i zastępowanie tekstu	54
2.6. Wyszukiwanie i zastępowanie tekstu bez uwzględniania wielkości liter	55
2.7. Tworzenie wyrażeń regularnych w celu uzyskania najkrótszego dopasowania	56
2.8. Tworzenie wyrażeń regularnych dopasowywanych do wielowierszowych wzorców	57
2.9. Przekształcanie tekstu w formacie Unicode na postać standardową	58
2.10. Używanie znaków Unicode w wyrażeniach regularnych	60
2.11. Usuwanie niepożądanych znaków z łańcuchów	61
2.12. Zapewnianie poprawności i porządkowanie tekstu	62
2.13. Wyrównywanie łańcuchów znaków	64
2.14. Łączenie łańcuchów znaków	66
2.15. Podstawianie wartości za zmienne w łańcuchach znaków	68
2.16. Formatowanie tekstu w celu uzyskania określonej liczby kolumn	70
2.17. Obsługiwanie encji HTML-a i XML-a w tekście	71
2.18. Podział tekstu na tokeny	73
2.19. Tworzenie prostego rekurencyjnego parsera zstępującego	75
2.20. Przeprowadzanie operacji tekstowych na łańcuchach bajtów	83
3. Liczby, daty i czas	87
3.1. Zaokrąglanie liczb	87
3.2. Przeprowadzanie dokładnych obliczeń na liczbach dziesiętnych	88
3.3. Formatowanie liczb w celu ich wyświetlenia	90
3.4. Stosowanie dwójkowych, ósemkowych i szesnastkowych liczb całkowitych	92
3.5. Pakowanie do bajtów i wypakowywanie z bajtów dużych liczb całkowitych	93
3.6. Przeprowadzanie obliczeń na liczbach zespolonych	95
3.7. Nieskończoność i wartości NaN	96
3.8. Obliczenia z wykorzystaniem ułamków	98
3.9. Obliczenia z wykorzystaniem dużych tablic liczbowych	99
3.10. Przeprowadzanie operacji na macierzach i z zakresu algebry liniowej	102
3.11. Losowe pobieranie elementów	103
3.12. Przekształcanie dni na sekundy i inne podstawowe konwersje związane z czasem	105
3.13. Określanie daty ostatniego piątku	107
3.14. Określanie przedziału dat odpowiadającego bieżącemu miesiącowi	108
3.15. Przekształcanie łańcuchów znaków na obiekty typu datetime	110
3.16. Manipulowanie datami z uwzględnieniem stref czasowych	111
4. Iteratory i generatory	113
4.1. Ręczne korzystanie z iteratora	113
4.2. Delegowanie procesu iterowania	114
4.3. Tworzenie nowych wzorców iterowania z wykorzystaniem generatorów	115
4.4. Implementowanie protokołu iteratora	117
4.5. Iterowanie w odwrotnej kolejności	119

4.6. Definiowanie funkcji generatorów z dodatkowym stanem	120
4.7. Pobieranie wycinków danych zwracanych przez iterator	121
4.8. Pomijanie pierwszej części obiektu iterowalnego	122
4.9. Iterowanie po wszystkich możliwych kombinacjach lub permutacjach	124
4.10. Przechodzenie po parach indeks – wartość sekwencji	125
4.11. Jednoczesne przechodzenie po wielu sekwencjach	127
4.12. Przechodzenie po elementach z odrębnych kontenerów	129
4.13. Tworzenie potoków przetwarzania danych	130
4.14. Przekształcanie zagnieżdżonych sekwencji na postać jednowymiarową	133
4.15. Przechodzenie po scalonych posortowanych obiektach iterowalnych zgodnie z kolejnością sortowania	134
4.16. Zastępowanie nieskończonych pętli while iteratorem	135
5. Pliki i operacje wejścia-wyjścia	137
5.1. Odczyt i zapis danych tekstowych	137
5.2. Zapisywanie danych z funkcji print() do pliku	139
5.3. Stosowanie niestandardowych separatorów lub końca wiersza w funkcji print()	140
5.4. Odczyt i zapis danych binarnych	141
5.5. Zapis danych do pliku, który nie istnieje	142
5.6. Wykonywanie operacji wejścia-wyjścia na łańcuchach	143
5.7. Odczytywanie i zapisywanie skompresowanych plików z danymi	144
5.8. Przechodzenie po rekordach o stałej wielkości	145
5.9. Wczytywanie danych binarnych do zmiennego bufora	146
5.10. Odwzorowywanie plików binarnych w pamięci	148
5.11. Manipulowanie ścieżkami	150
5.12. Sprawdzanie, czy plik istnieje	151
5.13. Pobieranie listy zawartości katalogu	152
5.14. Nieuwzględnianie kodowania nazw plików	153
5.15. Wyświetlanie nieprawidłowych nazw plików	154
5.16. Dodawanie lub zmienianie kodowania otwartego pliku	156
5.17. Zapisywanie bajtów w pliku tekstowym	158
5.18. Umieszczanie deskryptora istniejącego pliku w obiekcie pliku	159
5.19. Tworzenie tymczasowych plików i katalogów	160
5.20. Komunikowanie z portami szeregowymi	162
5.21. Serializowanie obiektów Pythona	163
6. Kodowanie i przetwarzanie danych	167
6.1. Wczytywanie i zapisywanie danych CSV	167
6.2. Wczytywanie i zapisywanie danych w formacie JSON	170
6.3. Parsowanie prostych danych w XML-u	174
6.4. Stopniowe parsowanie bardzo dużych plików XML	176

6.5. Przekształcanie słowników na format XML	179
6.6. Parsowanie, modyfikowanie i ponowne zapisywanie dokumentów XML	181
6.7. Parsowanie dokumentów XML z przestrzeniami nazw	183
6.8. Komunikowanie się z relacyjnymi bazami danych	185
6.9. Dekodowanie i kodowanie cyfr w systemie szesnastkowym	187
6.10. Dekodowanie i kodowanie wartości w formacie Base64	188
6.11. Odczyt i zapis tablic binarnych zawierających struktury	188
6.12. Wczytywanie zagnieżdżonych struktur binarnych o zmiennej długości	192
6.13. Podsumowywanie danych i obliczanie statystyk	200
7. Funkcje	203
7.1. Pisanie funkcji przyjmujących dowolną liczbę argumentów	203
7.2. Tworzenie funkcji przyjmujących argumenty podawane wyłącznie za pomocą słów kluczowych	204
7.3. Dołączanie metadanych z informacjami do argumentów funkcji	205
7.4. Zwracanie wielu wartości przez funkcje	206
7.5. Definiowanie funkcji z argumentami domyślnymi	207
7.6. Definiowanie funkcji anonimowych (wewnątrzwerszowych)	210
7.7. Pobieranie wartości zmiennych w funkcjach anonimowych	211
7.8. Uruchamianie n-argumentowej jednostki wywoływalnej z mniejszą liczbą argumentów	212
7.9. Zastępowanie klas z jedną metodą funkcjami	215
7.10. Dodatkowy stan w funkcjach wywoływanych zwrotnie	216
7.11. Wewnątrzwerszowe zapisywanie wywoływanych zwrotnie funkcji	219
7.12. Dostęp do zmiennych zdefiniowanych w domknięciu	221
8. Klasy i obiekty	225
8.1. Modyfikowanie tekstowej reprezentacji obiektów	225
8.2. Modyfikowanie formatowania łańcuchów znaków	226
8.3. Dodawanie do obiektów obsługi protokołu zarządzania kontekstem	228
8.4. Zmniejszanie zużycia pamięci przy tworzeniu dużej liczby obiektów	230
8.5. Hermetyzowanie nazw w klasie	231
8.6. Tworzenie atrybutów zarządzanych	232
8.7. Wywoływanie metod klasy bazowej	236
8.8. Rozszerzanie właściwości w klasie pochodnej	240
8.9. Tworzenie nowego rodzaju atrybutów klasy lub egzemplarza	243
8.10. Stosowanie właściwości obliczanych w leniwy sposób	246
8.11. Upraszczenie procesu inicjowania struktur danych	248
8.12. Definiowanie interfejsu lub abstrakcyjnej klasy bazowej	251
8.13. Tworzenie modelu danych lub systemu typów	254

8.14. Tworzenie niestandardowych kontenerów	259
8.15. Delegowanie obsługi dostępu do atrybutów	262
8.16. Definiowanie więcej niż jednego konstruktora w klasie	266
8.17. Tworzenie obiektów bez wywoływania metody <code>__init__()</code>	267
8.18. Rozszerzanie klas za pomocą klas mieszanych	269
8.19. Implementowanie obiektów ze stanem lub maszyn stanowych	273
8.20. Wywoływanie metod obiektu na podstawie nazwy w łańcuchu znaków	278
8.21. Implementowanie wzorca odwiedzający	279
8.22. Implementowanie wzorca odwiedzający bez stosowania rekurencji	283
8.23. Zarządzanie pamięcią w cyklicznych strukturach danych	288
8.24. Tworzenie klas z obsługą porównań	291
8.25. Tworzenie obiektów zapisywanych w pamięci podręcznej	293
9. Metaprogramowanie	297
9.1. Tworzenie nakładek na funkcje	297
9.2. Zachowywanie metadanych funkcji przy pisaniu dekoratorów	299
9.3. Pobieranie pierwotnej funkcji z nakładki	300
9.4. Tworzenie dekoratorów przyjmujących argumenty	302
9.5. Definiowanie dekoratora z atrybutami dostosowywanymi przez użytkownika	303
9.6. Definiowanie dekoratorów przyjmujących opcjonalny argument	306
9.7. Wymuszanie sprawdzania typów w funkcji za pomocą dekoratora	307
9.8. Definiowanie dekoratorów jako elementów klasy	311
9.9. Definiowanie dekoratorów jako klas	312
9.10. Stosowanie dekoratorów do metod klasy i metod statycznych	315
9.11. Pisanie dekoratorów, które dodają argumenty do funkcji w nakładkach	316
9.12. Stosowanie dekoratorów do poprawiania definicji klas	319
9.13. Używanie metaklasz do kontrolowania tworzenia obiektów	320
9.14. Sprawdzanie kolejności definiowania atrybutów klasy	323
9.15. Definiowanie metaklasz przyjmujących argumenty opcjonalne	325
9.16. Sprawdzanie sygnatury na podstawie argumentów <code>*args</code> i <code>**kwargs</code>	327
9.17. Wymuszanie przestrzegania konwencji pisania kodu w klasie	330
9.18. Programowe definiowanie klas	332
9.19. Inicjowanie składowych klasy w miejscu definicji klasy	335
9.20. Przeciążanie metod z wykorzystaniem uwag do funkcji	337
9.21. Unikanie powtarzających się metod właściwości	342
9.22. Definiowanie w łatwy sposób menedżerów kontekstu	344
9.23. Wykonywanie kodu powodującego lokalne efekty uboczne	346
9.24. Parsowanie i analizowanie kodu źródłowego Pythona	348
9.25. Dezasemblacja kodu bajtowego Pythona	351

10. Moduły i pakiety	355
10.1. Tworzenie hierarchicznych pakietów z modułami	355
10.2. Kontrolowanie importowania wszystkich symboli	356
10.3. Importowanie modułów podrzędnych z pakietu za pomocą nazw względnych	357
10.4. Podział modułu na kilka plików	358
10.5. Tworzenie odrębnych katalogów z importowanym kodem z jednej przestrzeni nazw	361
10.6. Ponowne wczytywanie modułów	362
10.7. Umożliwianie wykonywania kodu z katalogu lub pliku zip jako głównego skryptu	364
10.8. Wczytywanie pliku z danymi z pakietu	365
10.9. Dodawanie katalogów do zmiennej sys.path	366
10.10. Importowanie modułów na podstawie nazwy z łańcucha znaków	367
10.11. Wczytywanie modułów ze zdalnego komputera z wykorzystaniem haków w poleceniu importu	368
10.12. Modyfikowanie modułów w trakcie importowania	382
10.13. Instalowanie pakietów tylko na własny użytek	384
10.14. Tworzenie nowego środowiska Pythona	385
10.15. Rozpowszechnianie pakietów	386
11. Sieci i rozwijanie aplikacji sieciowych	389
11.1. Interakcja z usługami HTTP za pomocą kodu klienta	389
11.2. Tworzenie serwera TCP	393
11.3. Tworzenie serwera UDP	395
11.4. Generowanie przedziałów adresów IP na podstawie adresu CIDR	397
11.5. Tworzenie prostego interfejsu opartego na architekturze REST	399
11.6. Obsługa prostych zdalnych wywołań procedur za pomocą protokołu XML-RPC	403
11.7. Prosta komunikacja między interpreterami	405
11.8. Implementowanie zdalnych wywołań procedur	407
11.9. Proste uwierzytelnianie klientów	410
11.10. Dodawanie obsługi protokołu SSL do usług sieciowych	412
11.11. Przekazywanie deskryptora pliku gniazda między procesami	417
11.12. Operacje wejścia-wyjścia sterowane zdarzeniami	422
11.13. Wysyłanie i odbieranie dużych tablic	427
12. Współbieżność	429
12.1. Uruchamianie i zatrzymywanie wątków	429
12.2. Ustalanie, czy wątek rozpoczął pracę	432
12.3. Komunikowanie się między wątkami	434
12.4. Blokowanie sekcji krytycznej	439
12.5. Blokowanie z unikaniem zakleszczenia	441
12.6. Zapisywanie stanu wątku	445

12.7. Tworzenie puli wątków	446
12.8. Proste programowanie równoległe	449
12.9. Jak radzić sobie z mechanizmem GIL (i przestać się nim martwić)	453
12.10. Definiowanie zadań działających jak aktry	456
12.11. Przesyłanie komunikatów w modelu publikuj-subskrybuj	459
12.12. Używanie generatorów zamiast wątków	462
12.13. Odpytywanie wielu kolejek wątków	468
12.14. Uruchamianie procesu demona w systemie Unix	471
13. Skrypty narzędziowe i zarządzanie systemem	475
13.1. Przyjmowanie danych wejściowych skryptu za pomocą przekierowań, potoków lub plików wejściowych	475
13.2. Kończenie pracy programu wyświetleniem komunikatu o błędzie	476
13.3. Parsowanie opcji z wiersza poleceń	477
13.4. Prośba o podanie hasła w czasie wykonywania programu	479
13.5. Pobieranie rozmiarów terminala	480
13.6. Wywoływanie zewnętrznych poleceń i pobieranie danych wyjściowych	481
13.7. Kopiowanie lub przenoszenie plików i katalogów	482
13.8. Tworzenie i wypakowywanie archiwów	484
13.9. Wyszukiwanie plików na podstawie nazwy	485
13.10. Wczytywanie plików konfiguracyjnych	486
13.11. Dodawanie mechanizmu rejestrowania operacji do prostych skryptów	489
13.12. Dodawanie obsługi rejestrowania do bibliotek	491
13.13. Tworzenie stopera	493
13.14. Określanie limitów wykorzystania pamięci i procesora	494
13.15. Uruchamianie przeglądarki internetowej	495
14. Testowanie, debugowanie i wyjątki	497
14.1. Testowanie danych wyjściowych wysyłanych do strumienia stdout	497
14.2. Podstawianie obiektów w testach jednostkowych	498
14.3. Sprawdzanie wystąpienia wyjątków w testach jednostkowych	501
14.4. Zapisywanie danych wyjściowych testu w pliku	503
14.5. Pomijanie testów lub przewidywanie ich niepowodzenia	504
14.6. Obsługa wielu wyjątków	505
14.7. Przechwytywanie wszystkich wyjątków	507
14.8. Tworzenie niestandardowych wyjątków	508
14.9. Zgłaszanie wyjątku w odpowiedzi na wystąpienie innego wyjątku	510
14.10. Ponowne zgłaszanie ostatniego wyjątku	512
14.11. Wyświetlanie komunikatów ostrzegawczych	513
14.12. Debugowanie prostych awarii programu	514
14.13. Profilowanie i pomiar czasu pracy programów	516
14.14. Przyspieszanie działania programów	518

15. Rozszerzenia w języku C	525
15.1. Dostęp do kodu w języku C za pomocą modułu ctypes	526
15.2. Pisanie prostych modułów rozszerzeń w języku C	532
15.3. Pisanie funkcji rozszerzeń manipulujących tablicami	535
15.4. Zarządzanie nieprzejrzyistymi wskaźnikami w modułach rozszerzeń w języku C	538
15.5. Definiowanie i eksportowanie interfejsów API języka C w modułach rozszerzeń	540
15.6. Wywoływanie kodu Pythona w kodzie w języku C	544
15.7. Zwalnianie blokady GIL w rozszerzeniach w języku C	548
15.8. Jednoczesne wykonywanie wątków z kodu w językach C i Python	549
15.9. Umieszczanie kodu w języku C w nakładkach opartych na narzędziu Swig	550
15.10. Używanie Cythona do tworzenia nakładek na istniejący kod w języku C	555
15.11. Używanie Cythona do pisania wydajnych operacji na tablicach	560
15.12. Przekształcanie wskaźnika do funkcji w jednostkę wywoływalną	564
15.13. Przekazywanie łańcuchów znaków zakończonych symbolem NULL do bibliotek języka C	565
15.14. Przekazywanie łańcuchów znaków Unicode do bibliotek języka C	569
15.15. Przekształcanie łańcuchów znaków z języka C na ich odpowiedniki z Pythona	573
15.16. Używanie łańcuchów znaków o nieznanym kodowaniu pobieranych z języka C	574
15.17. Przekazywanie nazw plików do rozszerzeń w języku C	577
15.18. Przekazywanie otwartych plików do rozszerzeń w języku C	578
15.19. Wczytywanie w języku C danych z obiektów podobnych do plików	579
15.20. Pobieranie obiektów iterowalnych w języku C	581
15.21. Diagnozowanie błędów segmentacji	582
A Dalsza lektura	585
Skorowidz	587

Testowanie, debugowanie i wyjątki

Testowanie jest ciekawe, natomiast z debugowaniem sytuacja wygląda inaczej. Ponieważ nie istnieje kompilator analizujący kod przed wykonaniem go przez Pythona, testy są niezbędną częścią procesu tworzenia oprogramowania. W tym rozdziale opisano pewne często występujące problemy związane z testowaniem, debugowaniem i obsługą wyjątków. Nie jest to jednak proste wprowadzenie do programowania sterowanego testami lub korzystania z modułu `unittest`. Zakładamy, że opanowałeś już podstawowe zagadnienia z obszaru testów.

14.1. Testowanie danych wyjściowych wysyłanych do strumienia `stdout`

Problem

W programie działa metoda, która zwraca dane wyjściowe do standardowego strumienia wyjścia (`sys.stdout`). Prawie zawsze oznacza to, że tekst jest wyświetlany na ekranie. Programista chce napisać test dowodzący, że dla poprawnych danych wejściowych wyświetlane są odpowiednie dane wyjściowe.

Rozwiązanie

Za pomocą funkcji `patch()` modułu `unittest.mock` można stosunkowo łatwo zasymulować działanie strumienia `sys.stdout` w jednym teście, a następnie usunąć używany do tego obiekt. Pozwala to uniknąć stosowania kłopotliwych zmiennych tymczasowych i wyciekania symulowanego stanu między wykonywaniem poszczególnych testów.

Przyjrzyj się przykładowej funkcji z modułu `mymodule`:

```
# mymodule.py

def urlprint(protocol, host, domain):
    url = '{}://{}.{}'.format(protocol, host, domain)
    print(url)
```

Wbudowana funkcja `print` domyślnie wysyła dane wyjściowe do strumienia `sys.stdout`. Aby sprawdzić, czy dane wyjściowe rzeczywiście trafiają do tego strumienia, można zasymulować jego działanie za pomocą obiektu zastępczego, a następnie wykorzystać asercje dotyczące

wykonanych operacji. Metoda `patch()` modułu `unittest.mock` pozwala na wygodne zastępowanie obiektów w kontekście działającego testu. Natychmiast po zakończeniu testu przywracany jest pierwotny stan programu. Oto kod testujący moduł `mymodule`:

```
from io import StringIO
from unittest import TestCase
from unittest.mock import patch
import mymodule

class TestURLPrint(TestCase):
    def test_url_gets_to_stdout(self):
        protocol = 'http'
        host = 'www'
        domain = 'example.com'
        expected_url = '{}://{}.{}\n'.format(protocol, host, domain)

        with patch('sys.stdout', new=StringIO()) as fake_out:
            mymodule.urlprint(protocol, host, domain)
            self.assertEqual(fake_out.getvalue(), expected_url)
```

Omówienie

Funkcja `urlprint()` przyjmuje trzy argumenty, a test rozpoczyna się od podania fikcyjnych wartości każdego z nich. Zmienna `expected_url` jest ustawiana na łańcuch znaków z oczekiwanymi danymi wyjściowymi.

Aby uruchomić test, należy wykorzystać funkcję `unittest.mock.patch()` jako menedżera kontekstu w celu zastąpienia wartości `sys.stdout` obiektem typu `StringIO`. Zmienna `fake_out` to tworzony w tym procesie obiekt zastępczy. Można go wykorzystać w poleceniu `with` do przeprowadzenia różnych testów. Gdy polecenie `with` kończy pracę, funkcja `patch()` w wygodny dla programisty sposób przywraca wszystkie elementy do stanu przed uruchomieniem testu.

Warto zauważyć, że w Pythonie niektóre rozszerzenia w języku C mogą zapisywać dane bezpośrednio do standardowego wyjścia (z pominięciem strumienia `sys.stdout`). Ta receptura nie pozwala testować kodu opartego na takich rozszerzeniach, natomiast powinna działać poprawnie dla kodu napisanego w samym Pythonie. Jeśli chcesz przechwytywać operacje wejścia-wyjścia z rozszerzeń w języku C, możesz otworzyć tymczasowy plik i zastosować sztuczki związane z deskryptorami plików, aby czasowo przekierowywać do tego pliku dane ze standardowego wyjścia.

Więcej informacji na temat przechwytywania operacji wejścia-wyjścia związanych z łańcuchami znaków i obiektami typu `StringIO` znajdziesz w recepturze 5.6.

14.2. Podstawianie obiektów w testach jednostkowych

Problem

Programista pisze testy jednostkowe i chce podstawić wybrane obiekty, aby zastosować asercje związane z wykorzystaniem tych obiektów w czasie testów. Asercje te mogą dotyczyć wywołań z różnymi parametrami, dostępu do określonych atrybutów itd.

Rozwiązanie

W rozwiązaniu tego problemu pomocna będzie funkcja `unittest.mock.patch()`. Można jej użyć jako dekoratora (choć jest to dość nietypowe rozwiązanie), menedżera kontekstu lub niezależną funkcję. Oto przykładowy kod, w którym użyto jej jako dekoratora:

```
from unittest.mock import patch
import example

@patch('example.func')
def test1(x, mock_func):
    example.func(x)      # Wywołanie podstawionej funkcji example.func
    mock_func.assert_called_with(x)
```

Funkcję `patch()` można też wykorzystać jako menedżera kontekstu:

```
with patch('example.func') as mock_func:
    example.func(x)      # Wywołanie podstawionej funkcji example.func
    mock_func.assert_called_with(x)
```

Ponadto można ją zastosować do ręcznego podstawiania elementów kodu:

```
p = patch('example.func')
mock_func = p.start()
example.func(x)
mock_func.assert_called_with(x)
p.stop()
```

W razie potrzeby można połączyć dekoratory i menedżery kontekstu, aby podstawić grupę obiektów. Oto przykład:

```
@patch('example.func1')
@patch('example.func2')
@patch('example.func3')
def test1(mock1, mock2, mock3):
    ...

def test2():
    with patch('example.patch1') as mock1,\
        patch('example.patch2') as mock2,\
        patch('example.patch3') as mock3:
        ...
```

Omówienie

Funkcja `patch()` przyjmuje istniejący obiekt podany za pomocą pełnej nazwy i zastępuje go nową wartością. Pierwotna wartość jest przywracana po zakończeniu pracy funkcji z dekoratorem lub pracy menedżera kontekstu. Domyślnie wartości są zastępowane obiektami typu `MagicMock`:

```
>>> x = 42
>>> with patch('__main__.x'):
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

Można jednak zastąpić wartość czymś innym, podając zastępczy element jako drugi argument funkcji `patch()`:

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

Obiekty typu `MagicMock`, standardowo używane jako wartości zastępcze, mają naśladować pracę jednostek wywoływalnych i obiektów. Rejestrują informacje na temat używania i pozwalają stosować asercje. Oto przykład:

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>
>>> m.upper.return_value = 'WITAJ'
>>> m.upper('witaj')
'WITAJ'
>>> assert m.upper.called

>>> m.split.return_value = ['witaj', 'świecie']
>>> m.split('witaj świecie')
['witaj', 'świecie']
>>> m.split.assert_called_with('witaj świecie')
>>>

>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>
```

Zwykle operacje tego rodzaju przeprowadza się w testach jednostkowych. Załóżmy, że używasz następującej funkcji:

```
# example.py
from urllib.request import urlopen
import csv

def dowprices():
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=s1l1')
    lines = (line.decode('utf-8') for line in u)
    rows = (row for row in csv.reader(lines) if len(row) == 2)
    prices = { name:float(price) for name, price in rows }
    return prices
```

Przy pobieraniu danych z internetu i ich parsowaniu funkcja ta używa standardowo wywołania `urlopen()`. W testach jednostkowych możesz jednak udostępnić samodzielnie opracowany przewidywalny zbiór danych. Oto przykład oparty na podstawianiu danych:

```
import unittest
from unittest.mock import patch
import io
import example

sample_data = io.BytesIO(b'''\
"IBM",91.1\r
"AA",13.25\r
"MSFT",27.72\r
\r
''')

class Tests(unittest.TestCase):
    @patch('example.urlopen', return_value=sample_data)
    def test_dowprices(self, mock_urlopen):
        p = example.dowprices()
        self.assertTrue(mock_urlopen.called)
        self.assertEqual(p,
            {'IBM': 91.1,
             'AA': 13.25,
             'MSFT': 27.72})

if __name__ == '__main__':
    unittest.main()
```

W tym kodzie funkcja `urlopen()` z modułu `example` jest zastępowana obiektem zastępczym, który zwraca obiekt `BytesIO()` zawierający przykładowe dane.

Ważnym, a przy tym trudnym do zauważenia aspektem przedstawionego testu jest to, że podstawiana jest funkcja `example.urlopen`, a nie `urllib.request.urlopen`. Przy podstawianiu elementów trzeba stosować nazwy w takiej postaci, w jakiej występują w testowanym kodzie. Ponieważ w przykładowym kodzie pojawia się polecenie `from urllib.request import urlopen`, funkcja `urlopen()` używana w funkcji `dowprices()` znajduje się w module `example`.

W tej recepturze przedstawiono tylko niewielką część możliwości modułu `unittest.mock`. Aby zapoznać się z bardziej zaawansowanymi mechanizmami, koniecznie zajrzyj do oficjalnej dokumentacji (<http://docs.python.org/3/library/unittest.mock>).

14.3. Sprawdzanie wystąpienia wyjątków w testach jednostkowych

Problem

Programista chce napisać testy jednostkowe, które w elegancki sposób sprawdzają, czy wystąpił wyjątek.

Rozwiązanie

Aby sprawdzić wystąpienie wyjątków, należy zastosować metodę `assertRaises()`. Jeśli chcesz ustalić, czy funkcja zgłosiła wyjątek `ValueError`, zastosuj następujący kod:

```

import unittest

# Prosta przykładowa funkcja
def parse_int(s):
    return int(s)

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaises(ValueError, parse_int, 'N/A')

```

Jeśli chcesz sprawdzić wartość wyjątku, musisz zastosować inne podejście. Oto przykład:

```

import errno

class TestIO(unittest.TestCase):
    def test_file_not_found(self):
        try:
            f = open('/file/not/found')
        except IOError as e:
            self.assertEqual(e.errno, errno.ENOENT)
        else:
            self.fail('Wyjątek IOError nie wystąpił')

```

Omówienie

Metoda `assertRaises()` umożliwia wygodne sprawdzenie wystąpienia wyjątku. Częstym błędem jest samodzielne próbowanie wykrywania wyjątków w pisanych testach. Oto przykład:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)

```

Problem z tym podejściem polega na tym, że łatwo jest zapomnieć o warunkach brzegowych, np. o sytuacji, gdy w kodzie w ogóle nie wystąpił wyjątek. Aby uwzględnić tę sytuację, trzeba dodać dodatkowy warunek:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
        else:
            self.fail('Wyjątek ValueError nie wystąpił')

```

Metoda `assertRaises()` uwzględnia takie sytuacje, dlatego należy korzystać właśnie z niej.

Wadą metody `assertRaises()` jest to, że nie umożliwia określenia wartości utworzonego obiektu wyjątku. Wartość tę trzeba sprawdzić ręcznie. Rozwiązaniem pośrednim jest zastosowanie metody `assertRaisesRegex()`, która umożliwia sprawdzenie, czy wyjątek wystąpił, a jednocześnie porównuje łańcuchową reprezentację wyjątku z wyrażeniem regularnym. Oto przykład:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaisesRegex(ValueError, 'invalid literal .*',
                               parse_int, 'N/A')

```


Mało znaną cechą metod `assertRaises()` i `assertRaisesRegex()` jest to, że można z nich korzystać jak z menedżerów kontekstu:

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        with self.assertRaisesRegex(ValueError, 'invalid literal .*'):
            r = parse_int('N/A')
```

Ta postać może okazać się przydatna, jeśli test obejmuje kilka etapów (np. etap konfiguracji) oprócz samego wywołania jednostki wywoływanej.

14.4. Zapisywanie danych wyjściowych testu w pliku

Problem

Programista chce, aby dane wyjściowe testu były zapisywane w pliku, a nie przekazywane do standardowego wyjścia.

Rozwiązanie

Często stosowaną techniką przeprowadzania testów jednostkowych jest umieszczanie krótkiego fragmentu kodu w końcowej części pliku z kodem testu:

```
import unittest

class MyTest(unittest.TestCase):
    ...

if __name__ == '__main__':
    unittest.main()
```

Dzięki temu plik z kodem testu jest wykonywalny i wyświetla wyniki przeprowadzenia testu w standardowym wyjściu. Jeśli chcesz przekierować dane wyjściowe gdzie indziej, rozwiń wywołanie `main()` i napisz własną funkcję `main()`:

```
import sys
def main(out=sys.stderr, verbosity=2):
    loader = unittest.TestLoader()
    suite = loader.loadTestsFromModule(sys.modules[__name__])
    unittest.TextTestRunner(out, verbosity=verbosity).run(suite)

if __name__ == '__main__':
    with open('testing.out', 'w') as f:
        main(f)
```

Omówienie

Ciekawym aspektem tej receptury jest nie tyle przekierowywanie wyników testu do pliku, co fakt, że wykonanie tego zadania pozwala zrozumieć działanie wewnętrznych mechanizmów modułu `unittest`.

Na podstawowym poziomie moduł `unittest` najpierw tworzy zestaw testów. Składa się on z różnych zdefiniowanych metod testowych. Po przygotowaniu zestawu wykonywane są testy wchodzące w jego skład.

Obie wymienione części testów jednostkowych są niezależne od siebie. Tworzony w rozwiązaniu obiekt typu `unittest.TestLoader` służy do przygotowywania zestawu testów. Metoda `loadTestsFromModule()` to jedna z kilku metod do wczytywania testów. Tu metoda ta szuka w module klas `TestCase` i wczytuje z nich metody testowe. Jeśli potrzebujesz większej kontroli, możesz wykorzystać metodę `loadTestsFromTestCase()` (nie jest używana w kodzie) do pobrania metod testowych z konkretnych klas pochodnych od klasy `TestCase`.

Klasa `TextTestRunner` to przykładowa klasa przeprowadzająca testy. Głównym jej zadaniem jest wykonanie testów z zestawu. Klasa ta jest powiązana z funkcją `unittest.main()`. Tu skonfigurowano niskopoziomowe aspekty tej klasy — określono plik na dane wyjściowe i zwiększono poziom szczegółowości rejestrowanych danych.

Choć ta receptura zawiera tylko kilka wierszy kodu, pomaga zrozumieć, jak dostosować działanie modułu `unittest` do własnych potrzeb. Aby zmienić sposób tworzenia zestawu testów, należy wykonać odpowiednie operacje za pomocą klasy `TestLoader`. Jeśli chcesz zmodyfikować sposób przeprowadzania testów, utwórz niestandardową klasę do uruchamiania testów, działającą podobnie jak `TextTestRunner`. Omawianie tych zagadnień wykracza poza zakres tej książki. Dokładny przegląd potrzebnych protokołów znajdziesz w dokumentacji modułu `unittest`.

14.5. Pomijanie testów lub przewidywanie ich niepowodzenia

Problem

Programista chce ignorować lub oznaczać w testach jednostkowych wybrane testy, które zgodnie z oczekiwaniami mają zakończyć się niepowodzeniem.

Rozwiązanie

Moduł `unittest` udostępnia dekoratory, które można zastosować do wybranych metod testowych w celu uzyskania kontroli nad ich przebiegiem. Oto przykład:

```
import unittest
import os
import platform

class Tests(unittest.TestCase):
    def test_0(self):
        self.assertTrue(True)

    @unittest.skip('Test pominięto')
    def test_1(self):
        self.fail('Powinien zakończyć się niepowodzeniem!')

    @unittest.skipIf(os.name=='posix', 'Nieobsługiwane w systemach uniksowych')
    def test_2(self):
        import winreg

    @unittest.skipUnless(platform.system() == 'Darwin', 'Test dla systemu Mac OS')
    def test_3(self):
        self.assertTrue(True)
```

```

    @unittest.expectedFailure
    def test_4(self):
        self.assertEqual(2+2, 5)

if __name__ == '__main__':
    unittest.main()

```

Jeśli uruchomisz ten test na komputerze z systemem Mac OS, otrzymasz następujące dane wyjściowe:

```

bash % python3 testsample.py -v
test_0 (__main__.Tests) ... ok
test_1 (__main__.Tests) ... skipped 'Test pominięto'
test_2 (__main__.Tests) ... skipped 'Nieobsługiwane w systemach uniksowych'
test_3 (__main__.Tests) ... ok
test_4 (__main__.Tests) ... expected failure
-----
Ran 5 tests in 0.002s

OK (skipped=2, expected failures=1)

```

Omówienie

Dekorator `skip()` umożliwia pominięcie testu, którego nie chcesz przeprowadzać. Dekoratory `skipIf()` i `skipUnless()` mogą być przydatne do pisania testów, które dotyczą tylko określonych systemów operacyjnych lub wersji Pythona albo zależą od innego oprogramowania. Za pomocą dekoratora `@expectedFailure` możesz oznaczyć testy, o których wiesz, że powinny zakończyć się niepowodzeniem (i nie chcesz, aby platforma testowa generowała dodatkowe informacje na temat tych testów).

Dekoratory przeznaczone do pomijania metod można też stosować do całych klas testowych. Oto przykład:

```

@unittest.skipUnless(platform.system() == 'Darwin', 'Testy tylko dla systemu Mac OS')
class DarwinTests(unittest.TestCase):
    ...

```

14.6. Obsługa wielu wyjątków

Problem

Dany fragment kodu może zgłaszać różne wyjątki. Programista chce uwzględnić wszystkie możliwe wyjątki bez tworzenia powtarzających się fragmentów lub długich, skomplikowanych bloków kodu.

Rozwiązanie

Jeśli do obsługi różnych wyjątków można wykorzystać jeden blok kodu, wyjątki można pogrupować za pomocą krotki:

```

try:
    client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
    client_obj.remove_url(url)

```

W tym przykładzie metoda `remove_url()` jest wywoływana, gdy wystąpi jeden z wymienionych wyjątków. Jeśli jeden z wyjątków trzeba obsłużyć inaczej, należy umieścić go w klauzuli `except`:

```
try:
    client_obj.get_url(url)
except (URLError, ValueError):
    client_obj.remove_url(url)
except SocketTimeout:
    client_obj.handle_url_timeout(url)
```

Wyjątki często są grupowane w hierarchie dziedziczenia. Wtedy wszystkie wyjątki można przechwytywać przy użyciu klasy bazowej. Zamiast pisać następujący kod:

```
try:
    f = open(filename)
except (FileNotFoundError, PermissionError):
    ...
```

można zastosować polecenie `except` w poniższy sposób:

```
try:
    f = open(filename)
except OSError:
    ...
```

To rozwiązanie działa, ponieważ `OSError` to klasa bazowa wspólna dla wyjątków `FileNotFoundError` i `PermissionError`.

Omówienie

Choć poniższa technika dotyczy nie tylko obsługi *wielu* wyjątków, warto zauważyć, że do obsługi zgłaszanych wyjątków można też wykorzystać słowo kluczowe `as`:

```
try:
    f = open(filename)
except OSError as e:
    if e.errno == errno.ENOENT:
        logger.error('Pliku nie znaleziono')
    elif e.errno == errno.EACCES:
        logger.error('Odmowa uprawnień')
    else:
        logger.error('Nieoczekiwany błąd: %d', e.errno)
```

W tym przykładzie zmienna `e` zawiera obiekt zgłaszający wyjątek `OSError`. Jest to przydatne, jeśli trzeba zbadać wyjątek (np. obsłużyć go na podstawie wartości dodatkowego kodu stanu).

Pamiętaj, że klauzule `except` są sprawdzane w kolejności występowania i wykonywany jest kod z pierwszej pasującej klauzuli. Nie jest to idealne rozwiązanie, jednak można łatwo utworzyć kod, w którym do wyjątku pasuje kilka klauzul `except`:

```
>>> f = open('Brak')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'Brak'
>>> try:
...     f = open('Brak')
... except OSError:
...     print('Niepowodzenie')
... except FileNotFoundError:
...     print('Pliku nie znaleziono')
...
Niepowodzenie
>>>
```

Klauzula `except FileNotFoundError` nie jest uruchamiana, ponieważ wyjątek `OSError` jest ogólniejszy, pasuje do wyjątku `FileNotFoundError` i znajduje się pierwszy na liście.

Oto wskazówka dotycząca debugowania — jeśli nie znasz hierarchii klas obejmującej dany wyjątek, możesz ją szybko wyświetlić, sprawdzając wartość atrybutu `__mro__` tego wyjątku:

```
>>> FileNotFoundError.__mro__
(<class 'FileNotFoundError'>, <class 'OSError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
>>>
```

W poleceniu `except` można podać dowolną z wymienionych klas aż do `BaseException`.

14.7. Przechwytywanie wszystkich wyjątków

Problem

Programista chce napisać kod, który przechwytuje wszystkie wyjątki.

Rozwiązanie

Aby przechwytywać wszystkie wyjątki, należy napisać blok obsługi wyjątków typu `Exception`:

```
try:
    ...
except Exception as e:
    ...
    log('Powód:', e)    # To ważne!
```

Ten kod przechwytuje wszystkie wyjątki oprócz `SystemExit`, `KeyboardInterrupt` i `GeneratorExit`. Jeśli chcesz przechwytywać także te wyjątki, zmień typ `Exception` na `BaseException`.

Omówienie

Przechwytywanie wszystkich wyjątków jest czasem stosowane jako ułatwienie przez programistów, którzy nie potrafią zapamiętać każdego wyjątku możliwego w skomplikowanej operacji. Dlatego jeśli nie zachowasz ostrożności, jest to świetny sposób na utworzenie kodu, którego debugowanie będzie bardzo trudne.

Jeżeli zdecydujesz się przechwytywać wszystkie wyjątki, koniecznie rejestruj lub wyświetlaj powód wystąpienia wyjątku (możesz zapisywać przyczynę w pliku dziennika, wyświetlać na ekranie komunikat o błędzie itd.). W przeciwnym razie będziesz miał poważne trudności z ustaleniem przyczyny problemów. Przyjrzyj się następującemu przykładowi:

```
def parse_int(s):
    try:
        n = int(v)
    except Exception:
        print("Nieudane parsowanie")
```

Jeśli uruchomisz tę funkcję, uzyskasz następujące dane:

```
>>> parse_int('n/a')
Nieudane parsowanie
>>> parse_int('42')
Nieudane parsowanie
>>>
```

Możesz się zastanawiać, dlaczego kod nie działa. Teraz założmy, że funkcja wygląda tak:

```
def parse_int(s):
    try:
        n = int(v)
    except Exception as e:
        print("Nieudane parsowanie")
        print('Powód:', e)
```

Tym razem uzyskasz następujące dane, informujące, że programista popełnił błąd:

```
>>> parse_int('42')
Nieudane parsowanie
Powód: global name 'v' is not defined
>>>
```

Zwykle lepiej jest obsługiwać wyjątki konkretnego typu. Jeśli jednak musisz przechwytywać je wszystkie, zapewnij sobie dobre informacje diagnostyczne lub przekaz wyjątek, aby nie utracić danych o przyczynie błędu.

14.8. Tworzenie niestandardowych wyjątków

Problem

Programista tworzy aplikację i chce umieścić niskopoziomowe wyjątki w niestandardowych, które będą przekazywały więcej informacji w kontekście programu.

Rozwiązanie

Tworzenie nowych wyjątków jest łatwe. Wystarczy zdefiniować je jako klasy pochodne od klasy `Exception` (lub od jednego z innych istniejących typów wyjątków, jeśli ma to więcej sensu). Np. w kodzie dotyczącym sieci można zdefiniować niestandardowe wyjątki w następujący sposób:

```
class NetworkError(Exception):
    pass

class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

class ProtocolError(NetworkError):
    pass
```

Użytkownicy mogą następnie korzystać z tych wyjątków w standardowy sposób. Oto przykład:

```
try:
    msg = s.recv()
except TimeoutError as e:
    ...
except ProtocolError as e:
    ...
```

Omówienie

Niestandardowe klasy wyjątków prawie zawsze powinny dziedziczyć po wbudowanej klasie `Exception` lub po lokalnie zdefiniowanej klasie wyjątku podstawowego, która sama jest pochodna od `Exception`. Choć wszystkie wyjątki dziedziczą też po klasie `BaseException`, nie należy jej stosować jako klasy bazowej dla nowych wyjątków. Klasa `BaseException` jest zarezerwowana dla wyjątków związanych z wyjściem z systemu (takich jak `KeyboardInterrupt` i `SystemExit`) oraz innych, które sygnalizują aplikacji, że ma zakończyć pracę. Dlatego wyjątki tego rodzaju nie są przeznaczone do przechwytywania. Jeśli zastosujesz się do tej konwencji i wykorzystasz klasę `BaseException` jako bazową, niestandardowe wyjątki nie będą przechwytywane — zostaną uznane za sygnał natychmiastowego zamknięcia aplikacji!

Tworzenie w aplikacji niestandardowych wyjątków i stosowanie ich w przedstawiony sposób sprawia, że kod jest bardziej zrozumiały dla jego czytelników. W trakcie projektowania takiego kodu należy zastanowić się nad pogrupowaniem niestandardowych wyjątków za pomocą dziedziczenia. W skomplikowanych aplikacjach sensowne może być utworzenie dodatkowych klas bazowych, łączących różne klasy wyjątków. Dzięki temu użytkownik może przechwytywać ściśle określone błędy:

```
try:
    s.send(msg)
except ProtocolError:
    ...
```

a także bardziej ogólne grupy błędów:

```
try:
    s.send(msg)
except NetworkError:
    ...
```

Jeśli chcesz zdefiniować nowy wyjątek przesłaniający metodę `__init__()` z klasy `Exception`, koniecznie dodaj wywołanie `Exception.__init__()` ze wszystkimi przekazanymi argumentami. Oto przykład:

```
class CustomError(Exception):
    def __init__(self, message, status):
        super().__init__(message, status)
        self.message = message
        self.status = status
```

Może wygląda to dziwnie, jednak klasa `Exception` domyślnie przyjmuje wszystkie przekazane argumenty i zapisuje je w postaci krotki w atrybucie `.args`. Różne inne biblioteki i elementy Pythona działają tak, jakby wszystkie wyjątki udostępniały atrybut `.args`, dlatego jeśli pominiemy wspomniany krok, może się okazać, że w pewnych sytuacjach nowy wyjątek nie będzie działał poprawnie. Aby zrozumieć, jak stosować atrybut `.args`, przyjrzyj się poniższej interaktywnej sesji, w której wykorzystano wbudowany wyjątek `RuntimeError`. Zwróć uwagę na to, że w poleceniu `raise` można podać dowolną liczbę argumentów:

```
>>> try:
...     raise RuntimeError('Niepowodzenie')
... except RuntimeError as e:
...     print(e.args)
...
('Niepowodzenie',)
>>> try:
```

```
...     raise RuntimeError('Niepowodzenie', 42, 'spam')
... except RuntimeError as e:
...     print(e.args)
...
('Niepowodzenie', 42, 'spam')
>>>
```

Więcej informacji na temat tworzenia własnych wyjątków znajdziesz w dokumentacji Pythona (<http://docs.python.org/3/tutorial/errors.html>).

14.9. Zgłaszanie wyjątku w odpowiedzi na wystąpienie innego wyjątku

Problem

Programista zamierza zgłaszać wyjątek w odpowiedzi na przechwycenie innego wyjątku. Chce przy tym zapisywać w śladzie błędu informacje o obu wyjątkach.

Rozwiązanie

Aby połączyć wyjątki w łańcuch, zastosuj polecenie `raise from` zamiast prostego wywołania `raise`. Dzięki temu otrzymasz informacje o obu błędach. Oto przykład:

```
>>> def example():
...     try:
...         int('Brak')
...     except ValueError as e:
...         raise RuntimeError('Błąd parsowania') from e...
>>>
example()
Traceback (most recent call last):
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'Brak'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example
RuntimeError: Błąd parsowania
>>>
```

W śladzie błędu widać, że przechwytywane są oba wyjątki. Aby przechwycić wyjątek, należy zastosować standardowe polecenie `except`. Dodatkowo można jednak sprawdzić wartość atrybutu `__cause__` obiektu wyjątku, aby w razie potrzeby ustalić wyjątki z ich łańcucha. Oto przykład:

```
try:
    example()
except RuntimeError as e:
    print("Nie zadziałało:", e)
if e.__cause__:
    print('Powód:', e.__cause__)
```


Łańcuch wyjątków powstaje bez ingerencji programisty, gdy w bloku `except` zostaje zgłoszony inny wyjątek:

```
>>> def example2():
...     try:
...         int('Brak')
...     except ValueError as e:
...         print("Nieudane parsowanie:", err)
...
>>>
>>> example2()
Traceback (most recent call last):
  File "<stdin>", line 3, in example2
ValueError: invalid literal for int() with base 10: 'Brak'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example2
NameError: global name 'err' is not defined
>>>
```

W tym przykładzie dostępne są dane na temat obu wyjątków, jednak ich interpretacja jest odmienna. Tu wyjątek `NameError` jest zgłaszany w wyniku błędu programisty, a nie bezpośrednio w odpowiedzi na błąd parsowania. W takiej sytuacji atrybut `__cause__` wyjątku nie jest ustawiany. Zamiast tego atrybut `__context__` zostaje ustawiony na poprzedni wyjątek.

Jeśli z jakichś powodów nie chcesz łączyć wyjątków w łańcuch, zastosuj polecenie `raise from None`:

```
>>> def example3():
...     try:
...         int('Brak')
...     except ValueError:
...         raise RuntimeError('Błąd parsowania') from None...
>>>
example3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example3
RuntimeError: Błąd parsowania
>>>
```

Omówienie

W trakcie projektowania kodu należy zwrócić uwagę na stosowanie polecenia `raise` w innych blokach `except`. Zwykle takie polecenie należy zmienić na `raise from`. Preferowany powinien być kod w następującej postaci:

```
try:
    ...
except SomeException as e:
    raise DifferentException() from e
```

Wynika to z tego, że taki kod automatycznie łączy w łańcuch przyczyny błędów. Wyjątek `DifferentException` jest tu zgłaszany bezpośrednio w odpowiedzi na wyjątek `SomeException`. Ta zależność jest bezpośrednio widoczna w śladzie błędu.

Jeśli piszesz kod w poniższej postaci, wyjątki też są łączone w łańcuch, jednak często nie wiadomo, czy powstał on celowo, czy w wyniku nieoczekiwanego błędu w kodzie:

```
try:
    ...
except SomeException:
    raise DifferentException()
```

Polecenie `raise from` pozwala jednoznacznie określić, że programista chciał zgłosić drugi wyjątek.

Staraj się unikać blokowania informacji o wyjątkach, jak zrobiono to w poprzednim przykładzie. Choć blokowanie komunikatów może prowadzić do powstawania krótszych śladów błędów, powoduje też usunięcie informacji, które mogą okazać się przydatne w trakcie debugowania. Często najlepiej jest zachować tak dużo danych, jak to tylko możliwe.

14.10. Ponowne zgłaszanie ostatniego wyjątku

Problem

Programista przechwycił wyjątek w bloku `except` i teraz chce go ponownie zgłosić.

Rozwiązanie

Wystarczy zastosować samo polecenie `raise`. Oto przykład:

```
>>> def example():
...     try:
...         int('Brak')
...     except ValueError:
...         print("Nie zadziałało")
...         raise
...
>>> example()
Nie zadziałało
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'Brak'
>>>
```

Omówienie

Ten problem powstaje, gdy w odpowiedzi na wystąpienie wyjątku musisz podjąć pewne działania (np. zarejestrować operację, wykonać zadania porządkujące), a następnie przekazać wyjątek dalej. Przedstawioną technikę bardzo często stosuje się w blokach obsługi przechwytyjących wszystkie wyjątki:

```
try:
    ...
except Exception as e:
    # Przetwarzanie informacji o wyjątku
    ...

    # Przekazywanie wyjątku
    raise
```

14.11. Wyświetlanie komunikatów ostrzegawczych

Problem

Programista chce, aby program wyświetlał komunikaty ostrzegawcze (np. o przestarzałych funkcjach lub problemach z działaniem).

Rozwiązanie

Aby wyświetlać komunikaty ostrzegawcze w programie, należy zastosować funkcję `warnings.warn()`:

```
import warnings

def func(x, y, logfile=None, debug=False):
    if logfile is not None:
        warnings.warn('Argument logfile jest przestarzały', DeprecationWarning)
    ...
```

Argumentami funkcji `warn()` są komunikat ostrzegawczy oraz klasa ostrzeżenia (zwykle jest to jedna z następujących klas: `UserWarning`, `DeprecationWarning`, `SyntaxWarning`, `RuntimeWarning`, `ResourceWarning` lub `FutureWarning`).

Obsługa ostrzeżeń zależy od tego, w jaki sposób uruchomiono interpreter i skonfigurowano inne ustawienia. Jeśli uruchomisz Pythona z opcją `-W all`, uzyskasz dane wyjściowe w następującej postaci:

```
bash % python3 -W all example.py
example.py:5: DeprecationWarning: Argument logfile jest przestarzały
  warnings.warn('Argument logfile jest przestarzały', DeprecationWarning)
```

Zwykle ostrzeżenia powodują tylko wyświetlenie komunikatów w standardowym strumieniu błędów. Jeśli chcesz przekształcać ostrzeżenia w wyjątki, zastosuj opcję `-W error`:

```
bash % python3 -W error example.py
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    func(2, 3, logfile='log.txt')
  File "example.py", line 5, in func
    warnings.warn('Argument logfile jest przestarzały', DeprecationWarning)
DeprecationWarning: Argument logfile jest przestarzały
bash %
```

Omówienie

Generowanie komunikatów ostrzegawczych to przydatna technika zarządzania oprogramowaniem i pomagania użytkownikom przy wystąpieniu problemów, które nie wymagają zgłaszania wyjątków. Jeśli np. chcesz zmienić działanie biblioteki lub platformy, możesz zacząć wyświetlać komunikaty ostrzegawcze w przeznaczonym do modyfikacji kodzie. Pozwala to zachować przez pewien czas zgodność ze starszą wersją oprogramowania. Możesz też ostrzegać użytkowników o problemach, jakie mogą wynikać ze sposobu, w jaki korzystają z danego narzędzia we własnym kodzie.

Oto przykład ilustrujący zastosowanie ostrzeżeń w jednej z wbudowanych bibliotek, która generuje komunikat ostrzegawczy przy usuwaniu otwartego pliku:

```
>>> import warnings
>>> warnings.simplefilter('always')
>>> f = open('/etc/passwd')
>>> del f
__main__:1: ResourceWarning: unclosed file <io.TextIOWrapper name='/etc/passwd'
mode='r' encoding='UTF-8'>
>>>
```

Domyślnie nie wszystkie komunikaty ostrzegawcze są wyświetlane. Pokazywanie komunikatów ostrzegawczych w Pythonie można kontrolować za pomocą opcji `-W`. Ustawienie `-W all` powoduje wyświetlanie wszystkich takich komunikatów, a wartość `-W ignore` sprawia, że wszystkie są ignorowane. Ustawienie `-W error` pozwala przekształcić wszystkie ostrzeżenia w wyjątki. Wyświetlanie ostrzeżeń można też kontrolować za pomocą funkcji `warnings.simplefilter()`, tak jak w ostatnim fragmencie kodu. Argument `always` powoduje, że pokazywane są wszystkie komunikaty ostrzegawcze, wartość `ignore` prowadzi do ignorowania ostrzeżeń, a ustawienie `error` skutkuje przekształcaniem ostrzeżeń w wyjątki.

W prostych sytuacjach to wystarczy do generowania komunikatów ostrzegawczych. Moduł `warnings` udostępnia różne inne zaawansowane opcje konfiguracyjne związane z filtrowaniem i obsługą komunikatów ostrzegawczych. Więcej informacji na ten temat znajdziesz w dokumentacji Pythona (<http://docs.python.org/3/library/warnings.html>).

14.12. Debugowanie prostych awarii programu

Problem

Program nie działa i programista szuka prostej techniki debugowania.

Rozwiązanie

Jeśli program w momencie wystąpienia awarii zgłasza wyjątek, warto uruchomić kod za pomocą składni `python3 -i program.py`. Opcja `-i` powoduje otwarcie interaktywnej powłoki po zakończeniu pracy programu. Następnie można zbadać jego środowisko. Załóżmy, że kod programu wygląda tak:

```
# sample.py

def func(n):
    return n + 10

func('Witaj')
```

Gdy zastosujesz opcję `python3 -i`, uzyskasz następujące informacje:

```
bash % python3 -i sample.py
Traceback (most recent call last):
  File "sample.py", line 6, in <module>
    func('Witaj')
  File "sample.py", line 4, in func
    return n + 10
TypeError: Can't convert 'int' object to str implicitly
>>> func(10)
20
>>>
```

Jeśli nie można znaleźć oczywistych błędów, następnym krokiem po awarii jest uruchomienie debugera Pythona:

```
>>> import pdb
>>> pdb.pm()
> sample.py(4)func()
-> return n + 10
(Pdb) w
  sample.py(6)<module>()
-> func('Witaj')
> sample.py(4)func()
-> return n + 10
(Pdb) print n
'Witaj'
(Pdb) q
>>>
```

Jeżeli kod działa w środowisku, w którym trudno jest uzyskać dostęp do interaktywnej powłoki (np. na serwerze), często można samodzielnie przechwytywać błędy i generować ślad błędu:

```
import traceback
import sys

try:
    func(arg)
except:
    print('**** WYSTĄPIŁ BŁĄD ****')
    traceback.print_exc(file=sys.stderr)
```

Jeśli problem dotyczy nie awarii programu, a zwracania nieprawidłowych wyników lub nieoczekiwanego działania, często dobrym rozwiązaniem jest umieszczenie kilku wywołań `print()` w odpowiednich miejscach. Z podejściem tym powiązanych jest kilka ciekawych technik. Funkcja `traceback.print_stack()` pozwala utworzyć ślad stosu programu z miejsca jej wywołania:

```
>>> def sample(n):
...     if n > 0:
...         sample(n-1)
...     else:
...         traceback.print_stack(file=sys.stderr)
...
>>> sample(5)
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 5, in sample
>>>
```

Inna możliwość to ręczne uruchomienie debugera w dowolnym miejscu programu. W tym celu należy wywołać polecenie `pdb.set_trace()`:

```
import pdb

def func(arg):
    ...
    pdb.set_trace()
    ...
```

Technika ta może być przydatna, jeśli chcesz zapoznać się z mechanizmami pracy dużego programu, przepływem sterowania lub argumentami funkcji. Po uruchomieniu debugera można np. sprawdzić wartości zmiennych za pomocą funkcji `print` lub uruchomić polecenie `w`, aby wyświetlić ślad stosu:

Omówienie

Nie komplikuj niepotrzebnie debugowania. Do rozwiązania prostych problemów często wystarczy umiejętność czytania śladów błędów programu (sam błąd wyświetlany jest zwykle w ostatnim wierszu śladu). Dobrym podejściem może okazać się także umieszczenie w kodzie kilku wywołań funkcji `print()`, jeśli dopiero piszesz program i chcesz go zdiagnozować (wystarczy pamiętać, aby potem usunąć takie wywołania).

Debugger często stosuje się do sprawdzania wartości zmiennych w funkcji, która spowodowała awarię. Dlatego warto wiedzieć, jak uruchomić debugger po awarii.

Polecenia `pdb.set_trace()` i podobne są przydatne, jeśli próbujesz zrozumieć bardzo skomplikowany program, w którym przepływ sterowania nie jest oczywisty. Program działa wtedy do miejsca wywołania polecenia `set_trace()` i natychmiast uruchamia debugger. Następnie można przystąpić do próby zrozumienia problemu.

Jeśli piszesz kod w Pythonie za pomocą środowiska IDE, udostępnia ono zwykle własny interfejs do obsługi debugowania, oparty na poleceniu `pdb` lub zastępujący je. Więcej informacji znajdziesz w podręczniku dotyczącym używanego środowiska IDE.

14.13. Profilowanie i pomiar czasu pracy programów

Problem

Programista chce ustalić, ile czasu zajmuje programowi wykonywanie poszczególnych operacji, i przeprowadzić pomiary czasu pracy kodu.

Rozwiązanie

Jeśli chcesz zmierzyć czas pracy całego programu, zwykle wystarczy zastosować uniksowe polecenie `time` lub podobne narzędzie:

```
bash % time python3 someprogram.py
real 0m13.937s
user 0m12.162s
sys 0m0.098s
bash %
```

Drugą skrajnością jest generowanie szczegółowego raportu na temat pracy programu. Służy do tego moduł `cProfile`:

```
bash % python3 -m cProfile someprogram.py
859647 function calls in 16.016 CPU seconds
```

Ordered by: standard name

nccalls	tottime	percall	cumtime	percall	filename:lineno(function)
263169	0.080	0.000	0.080	0.000	someprogram.py:16(frange)
513	0.001	0.000	0.002	0.000	someprogram.py:30(generate_mandel)
262656	0.194	0.000	15.295	0.000	someprogram.py:32(<genexpr>)
1	0.036	0.036	16.077	16.077	someprogram.py:4(<module>)
262144	15.021	0.000	15.021	0.000	someprogram.py:4(in_mandelbrot)

```

1 0.000 0.000 0.000 0.000 os.py:746(urandom)
1 0.000 0.000 0.000 0.000 png.py:1056(_readable)
1 0.000 0.000 0.000 0.000 png.py:1073(Reader)
1 0.227 0.227 0.438 0.438 png.py:163(<module>)
512 0.010 0.000 0.010 0.000 png.py:200(group)
...
bash %

```

Zazwyczaj przy profilowaniu kodu stosuje się podejście pośrednie. Możliwe, że wiesz już, iż najwięcej czasu zajmuje programowi wykonywanie kilku określonych funkcji. Przy wybiórczym profilowaniu pracy funkcji przydatny może okazać się prosty dekorator:

```

# timethis.py

import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        r = func(*args, **kwargs)
        end = time.perf_counter()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper

```

Aby zastosować ten dekorator, wystarczy umieścić go przed definicją mierzonej funkcji. Oto przykład:

```

>>> @timethis
... def countdown(n):
...     while n > 0:
...         n -= 1
...
>>> countdown(10000000)
__main__.countdown : 0.803001880645752
>>>

```

Na potrzeby pomiaru czasu pracy bloku poleceń można zdefiniować menedżer kontekstu:

```

from contextlib import contextmanager

@contextmanager
def timeblock(label):
    start = time.perf_counter()
    try:
        yield
    finally:
        end = time.perf_counter()
        print('{} : {}'.format(label, end - start))

```

Oto przykład ilustrujący działanie tego menedżera kontekstu:

```

>>> with timeblock('Odliczanie'):
...     n = 10000000
...     while n > 0:
...         n -= 1
...
Odliczanie : 1.5551159381866455
>>>

```

Jeśli chcesz zbadać wydajność krótkich fragmentów kodu, przydatny będzie moduł `timeit`:

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>
```

Moduł `timeit` uruchamia milion razy polecenia podane w pierwszym argumencie i mierzy łączny czas ich wykonywania. Drugim argumentem jest konfiguracyjny łańcuch znaków, który jest uruchamiany w celu skonfigurowania środowiska przed przeprowadzeniem testu. Jeśli chcesz zmienić liczbę wykonań polecenia, podaj odpowiednią wartość w argumencie `number`:

```
>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>
```

Omówienie

Przy pomiarze wydajności należy pamiętać, że uzyskane wyniki nie są precyzyjne. Zastosowana w rozwiązaniu funkcja `time.perf_counter()` sprawia, że używany jest najdokładniejszy zegar w danym systemie. Jednak nawet ona mierzy czas zegarowy, dlatego wyniki zależą od wielu czynników, np. obciążenia komputera.

Jeśli interesuje Cię czas przetwarzania, a nie czas zegarowy, zastosuj funkcję `time.process_time()`:

```
from functools import wraps
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.process_time()
        r = func(*args, **kwargs)
        end = time.process_time()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper
```

Ponadto jeśli zamierzasz przeprowadzać dokładne analizy czasu pracy programu, koniecznie zapoznaj się z dokumentacją modułów `time`, `timeit` i pokrewnych, aby zrozumieć ważne różnice w ich działaniu w poszczególnych systemach operacyjnych, a także inne pułapki.

W recepturze 13.13 opisano powiązane zagadnienie — tworzenie klasy reprezentującej stoper.

14.14. Przyspieszanie działania programów

Problem

Program działa zbyt wolno i programista chce go przyspieszyć, nie stosując jednak skrajnych rozwiązań, takich jak rozszerzenia w języku C lub kompilator JIT.

Rozwiązanie

Jeśli za pierwszą zasadę optymalizacji uznać: „nie rób tego”, drugą prawie na pewno będzie: „nie optymalizuj mało istotnego kodu”. Dlatego jeśli program działa powoli, warto zacząć od profilowania kodu, co opisano w recepturze 14.13.

Zazwyczaj okazuje się, że większość czasu zajmuje wykonywanie kilku bloków kodu, np. wewnętrznych pętli przetwarzających dane. Po zidentyfikowaniu takich miejsc można wykorzystać proste techniki zaprezentowane w dalszych podpunktach, aby przyspieszyć pracę programu.

Stosowanie funkcji

Wielu programistów zaczyna stosować Pythona jako język do pisania prostych skryptów. W czasie tworzenia skryptów łatwo jest przyzwyczać się do pisania kodu o bardzo uproszczonej strukturze. Oto przykład:

```
# somescript.py

import sys
import csv

with open(sys.argv[1]) as f:
    for row in csv.reader(f):
        # Przetwarzanie danych
        ...
```

Mało znanym zjawiskiem jest to, że kod zdefiniowany w zasięgu globalnym (tak jak powyżej) działa wolniej od kodu umieszczonego w funkcji. Różnica w szybkości musi wynikać z zastosowania zmiennych lokalnych i globalnych (operacje z wykorzystaniem zmiennych lokalnych są szybsze). Dlatego jeśli chcesz przyspieszyć działanie programu, umieść polecenia skryptu w funkcji:

```
# somescript.py
import sys
import csv

def main(filename):
    with open(filename) as f:
        for row in csv.reader(f):
            # Przetwarzanie danych
            ...

main(sys.argv[1])
```

Wielkość różnicy w szybkości zależy od wykonywanych operacji. Jak wynika z naszego doświadczenia, przyspieszenie pracy o 15–30% nie jest niczym niezwykłym.

Wybiórcze eliminowanie operacji dostępu do atrybutów

Każde zastosowanie operatora kropki (.) w celu uzyskania dostępu do atrybutów związane jest z pewnymi kosztami. Na zapleczu wywoływane są wtedy metody specjalne, np. `__getattr__()` i `__getattribute__()`, co często prowadzi do wyszukiwania informacji w słowniku.

Często można uniknąć wyszukiwania atrybutów, importując je za pomocą polecenia `from module import name` i stosując metody powiązane. Przyjrzyj się następującemu fragmentowi kodu:

```
import math

def compute_roots(nums):
    result = []
    for n in nums:
        result.append(math.sqrt(n))
    return result

# Test
nums = range(1000000)
for n in range(100):
    r = compute_roots(nums)
```

Na naszym komputerze program ten wykonał się w około 40 sekund. Teraz zmodyfikuj funkcję `compute_roots()` w następujący sposób:

```
from math import sqrt

def compute_roots(nums):
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

Ta wersja kończy pracę w około 29 sekund. Jedyna różnica między wersjami polega na wyeliminowaniu operacji dostępu do atrybutu. Zamiast stosować polecenie `math.sqrt()`, wykorzystano polecenie `sqrt()`. Ponadto metodę `result.append()` zapisano w zmiennej lokalnej `result_append` i wykorzystano w wewnętrznej pętli.

Warto podkreślić, że zmiany te mają sens tylko w często wykonywanym kodzie, np. w pętlach. Dlatego stosowanie tej optymalizacji jest uzasadnione tylko w określonych miejscach.

Zmienne lokalne

Wcześniej wspomniano, że zmienne lokalne działają szybciej od globalnych. Jeśli kod często korzysta z danej nazwy, można przyspieszyć jego działanie, ograniczając zasięg zmiennej do jak najbardziej lokalnego. Przyjrzyj się zmodyfikowanej wersji opisanej wcześniej funkcji `compute_roots()`:

```
import math

def compute_roots(nums):
    sqrt = math.sqrt
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

W tej wersji metodę `sqrt` przeniesiono z modułu `math` do zmiennej lokalnej. Ta wersja kodu kończy pracę po około 25 sekundach (jest to poprawa w porównaniu z poprzednią wersją, której wykonanie zadania zajmowało 29 sekund). Dodatkowa poprawa wydajności wynika z tego, że lokalne wyszukiwanie zmiennej `sqrt` jest szybsze niż globalne wyszukiwanie jej odpowiednika.

Dostęp lokalny ma też znaczenie w klasach. Zwykle wyszukiwanie wartości za pomocą wywołania `self.name` jest wolniejsze niż dostęp do zmiennej lokalnej. W pętlach wewnętrznych czasem warto zapisać często używane atrybuty w zmiennych lokalnych. Oto przykład:

```
# Wolniejsza wersja
class SomeClass:
    ...
    def method(self):
        for x in s:
            op(self.value)

# Szybsza wersja
class SomeClass:
    ...
    def method(self):
        value = self.value
        for x in s:
            op(value)
```

Unikanie niepotrzebnej abstrakcji

Gdy dodajesz do kodu dodatkową warstwę operacji, np. za pomocą dekoratorów, właściwości lub deskryptorów, spowalniaasz pracę programu. Przyjrzyj się następującej klasie:

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, value):
        self._y = value
```

Teraz przeprowadź prosty pomiar czasu:

```
>>> from timeit import timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

Widać tu, że różnica w czasie dostępu do właściwości `y` i prostego atrybutu `x` jest duża — około 4,5-krotna. Jeśli ma to znaczenie, warto się zastanowić, czy definiowanie `y` jako właściwości jest konieczne. Jeżeli nie jest, należy z tego zrezygnować i zastosować prosty atrybut. To, że w programach pisanych w innych językach często wykorzystuje się funkcje do pobierania i ustawiania wartości, nie oznacza, że to samo podejście należy stosować w Pythonie.

Stosowanie wbudowanych kontenerów

Wbudowane typy danych (łańcuchy znaków, krotki, listy, zbiory i słowniki) są napisane w języku C i działają stosunkowo szybko. Jeśli chcesz tworzyć własne struktury danych (np. listy powiązane lub drzewa zrównoważone) zastępujące ich wbudowane odpowiedniki, uzyskanie podobnej wydajności może być trudne, a nawet niemożliwe. Dlatego często lepiej jest korzystać z wbudowanych typów danych.

Unikanie tworzenia niepotrzebnych struktur danych i kopii

Czasem programiści tworzą niepotrzebne struktury danych, gdy nie muszą wcale tego robić. Załóżmy, że programista napisał następujący kod:

```
values = [x for x in sequence]
squares = [x*x for x in values]
```

Możliwe, że najpierw chciał umieścić kolekcję wartości na liście, a następnie zastosować na nich operacje, np. wyrażenie listowe. Jednak pierwsza z tych list jest tu całkowicie zbędna. Wystarczy napisać kod w następującej postaci:

```
squares = [x*x for x in sequence]
```

Związane jest z tym inne zagadnienie — zwracaj uwagę na kod pisany przez programistów, którzy nadmiernie obawiają się charakterystycznego dla Pythona współużytkowania wartości. Nadużywanie funkcji `copy.deepcopy()` i podobnych wywołań może wskazywać na to, że autor kodu nie w pełni rozumie model zarządzania pamięcią w Pythonie lub nie ma do niego zaufania. Możliwe, że z takiego kodu można bezpiecznie usunąć wiele kopii.

Omówienie

Przed przystąpieniem do optymalizowania kodu zwykle warto najpierw przeanalizować zastosowane algorytmy. Znacznie większą poprawę wydajności można uzyskać, zmieniając wolny algorytm na jego odpowiednik o złożoności $O(n \log n)$, niż próbując poprawić implementację algorytmu o złożoności $O(n^2)$.

Jeśli już stwierdziłeś, że optymalizacja jest konieczna, popatrz na program z ogólnej perspektywy. Zwykle nie warto optymalizować każdej części programu, ponieważ kod stanie się wtedy nieczytelny i niezrozumiały. Zamiast tego skoncentruj się na fragmentach znacznie obniżających wydajność, np. na pętlach wewnętrznych.

Należy zachować ostrożność przy interpretowaniu wyników drobnych optymalizacji. Przyjrzyj się dwóm poniższym technikom tworzenia słownika:

```
a = {
    'name' : 'AAPL',
    'shares' : 100,
    'price' : 534.22
}

b = dict(name='AAPL', shares=100, price=534.22)
```

Druga wersja ułatwia pisanie, ponieważ nie trzeba podawać apostrofów wokół nazw kluczy. Jeśli jednak porównasz wydajność obu wywołań, przekonasz się, że wersja z poleceniem `dict()` jest trzy razy wolniejsza! Wiedząc to, możesz chcieć przejrzeć kod i zastąpić każde wywołanie `dict()` jego dłuższym odpowiednikiem. Jednak inteligentny programista koncentruje się tylko na tych fragmentach programu, w których optymalizacja ma znaczenie, np. w pętlach wewnętrznych. W innych miejscach różnica w szybkości będzie nieistotna.

Jeśli proste techniki przedstawione w tej recepturze nie pozwalają uzyskać pożądanej poprawy wydajności, możesz pomyśleć nad zastosowaniem rozwiązań opartych na kompilacji JIT. Projekt PyPy (<http://pypy.org/>) to implementacja interpretera Pythona, która analizuje działanie programu i generuje natywny kod maszynowy dla często wykonywanych fragmentów.

Czasem pozwala to przyspieszyć działanie programów w Pythonie o rząd wielkości. Dzięki temu pracują one prawie tak szybko (a czasem nawet szybciej) niż kod napisany w języku C. Niestety, wtedy gdy powstawała ta książka, interpreter PyPy nie obsługiwał w pełni Pythona 3. Możliwe jednak, że w przyszłości się to zmieni. Możesz też przyjrzeć się projektowi Numba (<http://numba.pydata.org/>). Numba to dynamiczny kompilator. Programista powinien za pomocą dekoratora oznaczyć wybrane funkcje Pythona jako przeznaczone do optymalizacji. Funkcje te są następnie kompilowane do natywnego kodu maszynowego z wykorzystaniem maszyny wirtualnej LLVM (<http://llvm.org/>). Także to podejście pozwala znacznie poprawić wydajność kodu, jednak (podobnie jak w interpreterze PyPy) obsługa Pythona 3 nie jest na razie kompletna.

Ponadto przychodzą nam na myśl słowa Johna Ousterhouta: „Największą poprawę wydajności zapewnia przejście od nie działającego do działającego kodu”. Nie martw się o optymalizację do momentu, w którym będzie potrzebna. Zagwarantowanie, że program działa poprawnie, jest zwykle ważniejsze niż zapewnienie jego szybkiej pracy (przynajmniej początkowo).

A

- abstrakcyjne klasy bazowe, 251, 325
- adres
 - CIDR, 397
 - IP, 397
 - URL, 373, 389
- akcesory, 222, 303
- aktory, 456, 458
- algorytm, 15
- algorytm rekurencyjny, 18
- alternatywa dla nakładek, 559
- aplikacja WSGI, 402
- aplikacje sieciowe, 389
- architektura REST, 389, 399, 401
- archiwa, 484
- argument
 - **kwargs, 298, 317, 327
 - *args, 298, 317, 329
 - debug, 318
- argumenty
 - domyślne funkcji, 207
 - funkcji, 203–206
 - z modyfikatorem *, 203
- ASCII, 63
- AST, abstract syntax tree, 348
- atrybut
 - __class__, 276, 277
 - __path__, 378
 - __slots__, 230, 250
 - __wrapped__, 301
 - ncalls, 314
 - request, 396
 - self.local, 445
- atrybuty zarządzane, 232
- automatyczne wczytywanie modułów, 368
- awaria interpretera, 529

B

- biblioteka
 - collections, 259
 - concurrent.futures, 449
 - csv, 167
 - ctypes, 455
 - distutils, 387
 - functools, 299
 - importlib, 381
 - LLVM, 565
 - logging, 460
 - multiprocessing, 407, 411
 - multiprocessing.connection, 406
 - numpy, 558
 - NumPy, 100, 102
 - optparse, 479
 - Pandas, 200
 - queue, 434
 - requests, 390
 - socket, 395
 - socketserver, 394, 413
 - somelib, 492
 - threading, 429–432, 439
 - urllib, 369, 392
 - xml.etree.ElementTree, 179
- blokada GIL, 431, 449, 453, 547, 557
- blokowanie
 - sekcji krytycznej, 439
 - z unikaniem zakleszczenia, 441
- błąd NameError, 346
- błędy
 - krytyczne, 582
 - segmentacji, 582
 - w kodowaniu Unicode, 576
- bufory wejścia-wyjścia, 579

C

certyfi­kat, 415
cykliczne struktury danych, 288
czas, 105
 pracy programu, 516
 UTC, 112

D

data, 105, 107
debugowanie, 370, 507, 512–516
definiowanie
 dekoratorów, 303, 306, 311
 konstruktorów w klasie, 266
 metaklas, 325
dekorator, 195, 219, 257, 291, 305
 @classmethod, 298, 315
 @contextmanager, 344
 @expectedFailure, 505
 @property, 298, 311
 @staticmethod, 298, 301, 315
 @typeassert, 308
 @when_imported, 383
 @wraps, 300
 skip(), 505
dekoratory
 bez argumentów, 306
 jako elementy klasy, 311
 jako klasa, 312
 poprawianie definicji klas, 319
 sprawdzanie typów, 307
 z argumentami opcjonalnymi, 306
 z atrybutami dostosowywanymi, 303
 zastosowanie do metod, 315
delegowanie
 obsługi dostępu, 262
 procesu iterowania, 114
 w klasach pośredniczących, 263
 zadań, 274
demony, 430, 471
deserializacja, 164
deskryptor, 159, 196, 243, 258, 418
dezasemblacja, 351
diagnozowanie błędów segmentacji, 582
dodawanie
 argumentów do sygnatur, 316
 elementów, 20
dokument
 Internet RFC 3875, 401
 PEP 302, 375, 381
 PEP 342, 468
 PEP 369, 384
 PEP 380, 468

 PEP 383, 576
 PEP 393, 570
 PEP 3118, 564
 PEP 3156, 468
 PEP 3333, 399
dokumentacja Cythona, 564
domknięcie, 216, 221, 343
dopasowanie
 najkrótsze wzorca, 56
 tekstu, 48–51
dostęp do
 atrybutów, 243, 262, 519
 kodu w języku C, 526
 zdalnych plików, 369
 zmiennych, 221
drzewo
 AST, 350
 parsowania, 78
dyspozytor, 402
dziedziczenie, 264
dzielenie
 łańcucha, 18, 47
 modułu, 358
 tekstu na tokeny, 73

E

eksportowanie interfejsów API, 540
element
 klucz, 27
 wartość, 27
ewaluator wyrażeń, 76, 81

F

filtrowanie
 elementów sekwencji, 37
 zawartości słownika, 28
flaga re.IGNORECASE, 55
format
 Base64, 188
 CSV, 167
 gzip, 449
 JSON, 170–172, 391, 409
 XML, 174, 179
formatowanie
 liczb, 90
 łańcuchów bajtów, 84
 łańcuchów znaków, 69, 226
 tekstu, 70
funkcja, 203
 append(), 182
 a2b_hex(), 187
 abort(), 546

acquire(), 442
 apply_async(), 220
 assertRaises(), 501
 atexit.register(), 474
 attrgetter(), 35
 avg(), 530
 bad_filename(), 156
 base64.b16encode(), 187
 bin(), 92
 bind(), 327
 bind_partial(), 309
 bz2.open(), 145
 calendar.monthrange(), 109
 center(), 64
 chain(), 129
 check_output(), 481, 482
 check_path(), 379
 collections.namedtuple(), 334
 combinations(), 125
 combining(), 60
 compile(), 369
 complex(), 95
 compress(), 39
 connect(), 185
 copytree(), 483
 countdown(), 432
 daemonize(), 473
 datetime.strptime(), 110
 dateutil.relativedelta(), 106
 defaultdict(), 37
 depth_first(), 117
 detach(), 156, 461
 dict(), 39
 dict.fromkeys(), 63
 dis(), 352
 divide(), 530
 endswith(), 49, 152
 enumerate(), 125
 exec(), 332, 346, 350
 fdopen(), 579
 fileinput.input(), 476
 fill(), 71
 filter(), 38
 find_loader(), 380
 find_module(), 376
 find_robots(), 450
 findall(), 52
 finditer(), 53
 float(), 96
 fnmatch(), 50
 fnmatchcase(), 50
 format(), 65, 90, 226
 format_map(), 68
 from_file(), 195
 from_list(), 530
 from_ndarray(), 530
 functools.wraps(), 313
 gen_concatenate(), 132
 generic_visit(), 282
 get(), 176
 get_archive_formats(), 484
 get_data(), 365
 get_exchange(), 459
 getattr(), 278
 getpass(), 480
 groupby(), 36
 gzip.open(), 145
 handle_receive(), 426
 handle_url(), 375
 heapq.heappop(), 21
 heapq.merge(), 134
 hmac.compare_digest(), 410
 html.escape(), 71
 imp.import_module(), 383
 imp.new_module(), 375
 imp.reload(), 362
 import_module(), 367
 importlib.import_module(), 367
 in_mandel(), 558
 indices(), 31
 insert(), 182
 int.bit_length(), 94
 int.from_bytes(), 93
 invalidate_caches(), 381
 islice(), 123
 itemgetter(), 34
 items(), 28
 iter(), 115, 136, 146
 iterparse(), 178
 iterparse(), 184
 itertools.chain(), 129
 itertools.combinations(), 124
 itertools.combinations_with_replacement(), 125
 itertools.dropwhile(), 122
 itertools.islice(), 123
 itertools.isslice(), 121
 itertools.zip_longest(), 128
 join(), 66, 436
 json.dump(), 171
 json.loads(), 172
 keys(), 28
 list(), 38
 ljust(), 65
 locals(), 347
 logged(), 302, 307
 main(), 503

funkcja

make_archive(), 484
makefile(), 160
match(), 52
match(), 53
math.fsum(), 90
math.isinf(), 97
math.isnan(), 97
memory_map(), 148
mkdtemp(), 161
mkstemp(), 162
mmap(), 149
most_common(), 31
NamedTemporaryFile(), 161
namedtuple(), 40
next(), 113
nlargest(), 20
normalize(), 59, 112
nsmallest(), 21
open(), 137–145, 159, 365
opener(), 216
operator.itemgetter(), 336
operator.methodcaller(), 279
os.get_terminal_size(), 480
os.listdir(), 152, 576
os.stat(), 153
os.walk(), 485
pack(), 190
pandas.read_csv(), 170
parse_end_remove(), 179
partial(), 212–215, 219
patch(), 497, 499
pickle.dumps(), 163
pickle.load(), 164
print(), 139
print_chars(), 569
print_result(), 217
Py_BuildValue(), 534
py_divide(), 534
PyArg_ParseTuple(), 534, 570
PyBuffer_GetBuffer(), 537
PyErr_Occurred(), 547
PyFile_FromFd(), 578
PyFloat_Check(), 547
PyGILState_Ensure(), 549
PyGILState_Release(), 548
PyInit_sample(), 535
PyIter_Next(), 582
PyObject_Call(), 580
PyObject_AsPoint(), 540
PyUnicode_AsWideCharString(), 572
q.empty(), 439
random.choice(), 103
random.randint(), 104
random.random(), 104
random.sample(), 103
random.seed(), 104
random.shuffle(), 104
range(), 109
re.compile(), 58
re.split(), 47
read(), 147, 580
read_polys(), 199
read_records(), 190
readinto(), 146
recv_handle(), 418
recvfrom(), 396
register_function(), 404
relativedelta(), 107
release(), 440
reload(), 363
remove(), 182
replace(), 62, 109
reversed(), 119
rjust(), 65
round(), 87
scanner(), 73
select(), 422
send(), 218, 287, 406, 460
send_handle(), 418
sendmsg(), 420
sendto(), 396
serve_forever(), 394, 404
setattr(), 250
setdefault(), 24
setrlimit(), 495
sig.bind(), 310
slice(), 31
socketpair(), 469
sort(), 213
sorted(), 27, 34
spam(), 239
split(), 47
ssl.RAND_bytes(), 105
ssl.wrap_socket(), 412
start_response(), 402
startswith(), 49, 152
str.endswith(), 48
str.replace(), 64
str.startswith(), 48
str.translate(), 62
strip(), 61
strptime(), 110
struct.unpack(), 193
struct.unpack_from(), 194
sub(), 55, 70
subprocess.check_output(), 481
super(), 236, 241, 256, 272
svc_login(), 480

- sys.getfilesystemencoding(), 153, 155
- task_done(), 436
- TemporaryDirectory(), 161
- TemporaryFile(), 161
- threading.local(), 445
- threading.stack_size(), 449
- throw(), 467
- time.perf_counter(), 518
- time.process_time(), 518
- time.time(), 494
- to os.path.normpath(), 486
- translate(), 62–64
- tuple(), 49
- types.new_class(), 332, 335
- unicodedata.normalize(), 62
- unittest.mock.patch(), 498
- unpack(), 190
- update(), 32, 45
- urlopen(), 369, 501
- urlprint(), 498
- values(), 27, 29
- vars(), 69
- visit(), 286
- warn(), 513
- warnings.simplefilter(), 514
- webbrowser.get(), 496
- wrapper(), 298
- xml.etree.ElementTree.parse(), 175
- zip(), 26, 127

funkcje

- anonimowe, 210
- argumenty, 203
- argumenty domyślne, 207
- fabryczne, 294
- interfejsu API, 542
- narzędziowe, 539, 541
- redukcyjne, 43
- rozszerzeń, 535
- słowa kluczowe, 204
- wywoływane zwrotnie, 216, 219
- zwracanie wartości, 206

G

- generator, 29, 116, 120, 462
 - adresów IP, 397
 - liczb losowych, 104
 - nakładek Swig, 550
- GIL, Global Interpreter Lock, 431, 449, 453
- gniazdo domeny, 420
- grupowanie rekordów, 35
- gwiazdka, 17, 18

H

- haki, 368
- hermetyzowanie nazw, 231

I

- implementowanie
 - obiektów, 273
 - protokołu iteratora, 117
 - wywołań zdalnych, 407
 - wzorca odwiedzającego, 279, 283
- importowanie
 - modułów, 357, 367
 - symboli, 356
- inicjowanie
 - składowych klasy, 335
 - struktur danych, 248
- instalowanie pakietów, 384
- interakcja
 - z bazą danych, 186
 - z usługami HTTP, 389
- interfejs, 251
 - API, 540
 - CGI, 401
 - IP, 398
 - Swiga, 551, 553
- iteracyjne przetwarzanie danych, 130
- iteratory, 113
- iterowalny obiekt, 15
- iterowanie w odwrotnej kolejności, 119

J

- jednoczesne wykonywanie wątków, 549
- jednostka wywoływalna, 564
- język C, 525
 - pobieranie obiektów iterowalnych, 581
 - pobieranie obiektów podobnych do plików, 579
 - przekształcanie łańcuchów znaków, 573
 - wywoływanie kodu Pythona, 544
- język Cython, 455, 555
- JSON, JavaScript Object Notation, 170

K

- kapsułki, 538, 539
- katalog
 - Scripts, 385
 - site-packages, 385
- katalogi instalacyjne, 384

- klasa, 225
 - Async, 219
 - BaseException, 509
 - BytesIO, 144
 - ChainMap, 44
 - collections.Counter, 31
 - Connection, 274
 - ConnectionState, 276
 - CountdownTask, 431
 - Descriptor, 256
 - DoubleArrayType, 530
 - EchoHandler, 214
 - Evaluator, 285
 - Exception, 508
 - ExpressionEvaluator, 78
 - FileInput, 476
 - LazyConnection, 229, 446
 - namedtuple, 40
 - NodeVisitor, 281
 - OSError, 506
 - ProcessPoolExecutor, 449, 451
 - Queue, 434
 - RLock, 440
 - RPCHandler, 408
 - RPCProxy, 408
 - Semaphore, 440
 - SimpleXMLRPCServer, 405
 - StreamRequestHandler, 393, 395
 - StringIO, 144
 - StructTupleMeta, 336
 - Structure, 194
 - subprocess.Popen, 482
 - TaskScheduler, 463
 - TCPServer, 214
 - TestLoader, 504
 - Thread, 431
 - Timer, 494
 - UDPServer, 396
 - UrlMetaFinder, 377, 381
 - UrlModuleLoader, 373
 - UrlPackageLoader, 378
 - UrlTemplate, 216
 - XMLNamespaces, 184
- klasy
 - bazowe, 236, 256
 - mieszane, 269, 271
 - pochodne, 240
 - pośredniczące, 263
- klauzula except FileNotFoundError, 507
- klient
 - HTTP, 392
 - TCP, 423
- klucz, 27
- klucz prywatny, 416
- kod
 - klienta, 419, 423
 - procesu roboczego, 421
 - Pythona w kodzie C, 544, 546
 - serwera, 423
 - w języku C, 525
- kodowanie
 - ASCII, 64
 - Base64, 188
 - cyfr szesnastkowych, 187
 - formatu CSV, 169
 - nazw plików, 153
 - otwartego pliku, 156
 - Unicode, 58, 60, 156
 - utf-8, 138
- kolejka, 434
 - nieograniczona, 20
 - o stałej długości, 19
 - priorytetowa, 22
- kolejki wątków, 468
- kolejność
 - definiowania atrybutów, 323
 - elementów, 25, 29
 - tokenów, 74
- kompresja
 - bz2, 145
 - gzip, 144
- komunikacja
 - między interpreterami, 405
 - między wątkami, 23, 435
- komunikat
 - o błędzie, 476
 - ostrzegawczy, 513
- konfigurowanie metaklasy, 327
- konsolidator języka C, 540
- konstruktor, 266
- kontrolowanie
 - definicji klas, 332
 - dekoratora, 303
 - importowania symboli, 356
 - tworzenia obiektów, 320
- konwencje pisania kodu, 330
- konwersje czasu, 105
- kopiec, 21
- kopiowanie
 - katalogów, 483
 - metadanych, 300
 - plików, 482
- krotki typu namedtuple, 41

L

leniwe obliczanie właściwości, 246
liczba argumentów, 212
liczby, 87

- całkowite, 92
- zespólone, 95
- zmiennoprzecinkowe, 88

limit wykorzystania pamięci, 494
lista, 18, 24

- `_post_import_hooks`, 383
- MRO, 238, 272
- `sys.meta_path`, 376
- `sys.path`, 378
- `sys.path_importer_cache`, 378

listy

- plików, 152
- słowników, 33

losowe pobieranie elementów, 103

Ł

łańcuch

- formatowanie, 69
- łączenie, 66
- obiekty typu `datetime`, 110
- podstawianie wartości, 68
- usuwanie znaków, 61

łańcuchy

- bajtów, 83, 93
- znaków, 47, 64, 180
 - nazwa metody, 278
 - nazwy modułów, 367
 - o nieznanym kodowaniu, 574

łączenie

- łańcuchów znaków, 66
- odwzorowań, 43
- wyjątków, 510

M

macierze, 102
maszyna

- stanowa, 273
- wirtualna LLVM, 523

menedżer kontekstu, 344
metadane, 152, 205, 299
metaklasy, 195, 200

- argumenty opcjonalne, 325
- kontrolowanie definicji, 332
- kontrolowanie tworzenia obiektów, 320

metaścieżki, 376
metoda, *Patrz także* funkcja

`__call__()`, 312
`__enter__()`, 228
`__exit__()`, 228
`__format__()`, 227
`__get__()`, 245, 247, 314
`__getattr__()`, 263
`__getattribute__`, 319
`__getstate__()`, 164
`__init__()`, 172, 248, 266, 328
`__iter__()`, 114, 259
`__missing__()`, 69
`__new__()`, 267, 294, 337
`__prepare__()`, 324, 334, 340
`__repr__()`, 225
`__reversed__()`, 119
`__setstate__()`, 164
`__str__()`, 226
`__complete()`, 426
`__get_links()`, 380
`__replace()`, 41

metody

- klasy bazowej, 236
- statyczne, 315
- właściwości, 342, 343

model

- aktorów, 456
- danych, 254
- publikuj-subskrybuj, 459

moduł, 355

- `argparse`, 477
- `ast`, 82
- `base64`, 187
- `binascii`, 187
- `cmath`, 95
- `collections`, 44, 191, 253, 261
- `concurrent.futures`, 425
- `configparser`, 486, 488
- `contextlib`, 344
- `csv`, 168, 169
- `ctypes`, 526, 528, 531
- `datetime`, 105, 107
- `dateutil`, 106
- `decimal`, 89
- `dis`, 352
- `ElementTree`, 176, 184
- `faulthandler`, 582
- `fileinput`, 475
- `fnmatch`, 50
- `fractions`, 98
- `getopt`, 479
- `getpass`, 479
- `heapq`, 20, 22
- `hmac`, 410

moduł
inspect, 327
io, 578
ipaddress, 397–399
itertools, 113, 122, 124
json, 170, 174
locale, 91
logging, 293, 489, 491
mmap, 149
multiprocessing, 213, 406, 431, 453
multiprocessing.connection, 405
numpy, 96
os.path, 150–152, 483
pickle, 163, 166, 406, 409
pytz, 111
random, 104
re, 55, 60
resource, 494
shutil, 482–484
sqlite3, 185
ssl, 412
string, 227
struct, 94, 189, 192, 537
subprocess, 482
tempfile, 160
textwrap, 70
time, 493
timeit, 518
unicodedata, 59
unittest, 504
unittest.mock, 497
urllib, 391
urllib.request, 389
warnings, 514
webbrowser, 496
xml.etree.ElementTree, 181
xml.sax.saxutils, 181
xmlrpc.client, 414

moduły rozszerzeń w języku C, 532

modyfikator
*, 203
**, 203

modyfikowanie
modułów, 382
słownika, 28

MRO, method resolution order, 238

N

nagłówki HTTP, 390
nakładki, 297, 550, 555
NaN, not a number, 96
narzędzia do parsowania, 81
narzędzie Swig, 550–554

nazwy
bezwzględne, 358
pakietów, 357
plików, 153, 154
stref czasowych, 112
wycinków, 30
względne, 357

nieskończoność, 96
normalizowanie, 59
normalizowanie ścieżki, 486

O

obiekt, 267

Actor, 457
ArgumentParser, 478
array, 530
bytes, 142
CFUNCTYPE, 565
ChainMap, 44
collections.deque, 19
Condition, 433
Connection, 275
Counter, 32
ctypes.c_int, 529
datetime, 105, 108, 110
Decimal, 89
defaultdict, 24
deque, 20
Element, 180
Event, 432
EventHandler, 423
IPv4Address, 398
Item, 23
Lock, 439
MagicMock, 500
mmap, 149
MultiMethod, 340
namedtuple, 41, 191
OrderedDict, 25, 324
Point, 540, 554
Profiled, 314
Queue, 435, 438
Result, 458
RLock, 440
Semaphore, 441
ServerProxy, 415
SizedRecord, 198
SortedItems, 260
Spam, 296
stdout, 160
Struct, 190
Structure, 199
SortedItems, 260

- TCPServer, 394
- ThreadPoolExecutor, 426, 447
- timedelta, 105, 107, 109
- UrlMetaFinder, 373
- UrlPathFinder, 380
- WeakValueDictionary, 295
- widoku elementów, 28
- widoku kluczy, 28
- obiekty
 - iterowalne, 16
 - podobne do plików, 579
- obliczenia
 - na tablicach, 99
 - na ułamkach, 98
- obsługa
 - gniazd, 428
 - iterowania, 117
 - łańcuchów znaków Unicode, 60, 569
 - nazw plików, 577
 - nieprzejrzystych struktur danych, 538
 - obiektów typu Point, 540
 - porównań, 291
 - protokołu SSL, 412, 417
 - protokołu zarządzania kontekstem, 228
 - rejestrowania, 491
 - synchronizacji, 440
 - wielu wyjątków, 505
 - wywołań zdalnych, 403
- odbieranie tablic, 427
- odczyt
 - danych binarnych, 141
 - danych tekstowych, 137
 - plików skompresowanych, 144
 - pliku tekstowego, 137
 - tablic binarnych, 188
- odpytywanie kolejek wątków, 468
- odwzorowywanie
 - kluczy, 24
 - nazw na elementy, 40
 - plików binarnych, 148
- ogranicznik rozdzielający pola, 48
- operacja przypisania, 15
- operacje
 - na łańcuchach bajtów, 83
 - na macierzach, 102
 - na tablicach, 560
 - skalarne, 99
 - wejścia-wyjścia, 137
 - wejścia-wyjścia na łańcuchach, 143
 - wejścia-wyjścia sterowane zdarzeniami, 422
- operator
 - %, 91
 - <, 109
 - is, 209

- operatory
 - matematyczne, 32
 - porównywania, 291, 293
- optymalizowanie kodu, 522

P

- pakiet Pandas, 170
- pakiety, 355
 - biblioteczne, 386
 - oparte na przestrzeni nazw, 361
- pamięć, 230, 288, 494
- pamięć podręczna, 293
- para klucz-wartość, 403
- parser
 - rekurencyjny, 79
 - zstępujący, 75
- parsowanie
 - danych, 174
 - dokumentu XML, 181–184
 - opcji, 477, 479
 - stopniowe, 176, 179
 - tekstu, 72
- pętla while, 135
- plik
 - logconfig.ini, 490
 - Makefile, 545
 - MANIFEST.in, 387
 - server_cert.pem, 417
 - server_key.pem, 416
 - setup.py, 387, 533, 542, 556
- pliki
 - .pth, 366
 - .pxd, 557
 - CSV, 167
 - konfiguracyjne, 486
 - tymczasowe, 160
- pobieranie
 - danych wyjściowych, 481
 - danych z internetu, 501
 - hasła, 479
 - katalogów, 152
 - listy plików, 152
 - rozmiarów terminala, 480
 - wartości zmiennych, 211
 - wycinków danych, 121
 - z dokumentu XML, 176
- podsumowania, 200
- podzbiór słownika, 39
- połączenie
 - goto, 546, 548
 - import, 368, 376
 - pyvenv, 385
 - raise, 511

- połączenie
 - raise from, 510
 - sudo, 385
 - time, 516
 - yield, 402, 467
 - połączenia wieloprocesowe, 419
 - pomiar wydajności, 518
 - pomijanie
 - początkowych elementów, 122
 - testów, 504
 - porównywanie krotek, 27
 - port szeregowy, 162
 - porządek bitów, 94
 - porządkowanie tekstu, 62
 - potokowe przekazywanie danych, 130, 475
 - powłoka, 481
 - priorytet elementu, 22
 - proces demona, 471
 - programowanie równoległe, 449
 - programowe definiowanie klas, 332
 - protokół
 - Oauth, 392
 - SSL, 410–413
 - UDP, 423
 - XML-RPC, 403, 404
 - przechodzenie
 - po elementach, 124–129
 - po obiektach posortowanych, 134
 - po rekordach, 145
 - przechwytywanie wszystkich wyjątków, 507
 - przeciążanie metod, 337
 - przedrostek 0o, 93
 - przekazywanie
 - deskryptora pliku, 417
 - łańcuchów do bibliotek, 565, 569
 - łańcuchów znaków Unicode, 573
 - nazw plików, 577
 - otwartych plików, 578
 - tablic, 536
 - przekierowywanie
 - pliku do skryptu, 475
 - wyników testu do pliku, 503
 - przekształcanie
 - dużych liczb, 94
 - łańcuchów znaków, 110, 573
 - na łańcuch bajtów, 180
 - słowników, 179
 - tekstu, 58
 - wartości, 534
 - zagnieżdżonych sekwencji, 133
 - przenoszenie plików, 482
 - przestrzenie nazw XML, 183
 - przesyłanie
 - datagramów, 396
 - komunikatów, 459
 - przetwarzanie
 - danych, 130, 167
 - dokumentów XML, 178
 - list, 18
 - tablic, 535, 560
 - tekstu, 47
 - przyspieszanie działania programów, 518
 - przywracanie pamięci, 290
 - pula wątków roboczych, 446
- ## R
- redukcja danych, 27, 42
 - rejestrowanie operacji, 489, 491
 - rekurencja, 19, 282
 - rekurencyjne parsery zstępujące, 80
 - relacyjne bazy danych, 185
 - rozmiary terminala, 480
 - rozpowszechnianie pakietów, 386
 - rozszerzanie
 - klas, 269
 - właściwości, 240
 - rozszerzenia w języku C, 525
 - rozszerzenie llvmpy, 565
 - RPC, remote procedure call, 407
- ## S
- sekwencja słowników, 35
 - sekwencje, 15
 - filtrowanie elementów, 37
 - najczęściej występujące elementy, 31
 - przekształcanie, 133
 - usuwanie powtórzeń, 29
 - semafory, 434
 - serializacja, 163, 165
 - serializowanie danych, 428
 - serwer
 - Apache, 401
 - TCP, 393, 423
 - UDP, 395
 - XML-RPC, 413
 - sieci, 389
 - skrypt, 364
 - przetwarzanie hasła, 480
 - rejestrowanie operacji, 489
 - uruchamianie przeglądarki, 495
 - skrypty narzędziowe, 475
 - słownik
 - MultiDict, 340
 - OrderedDict, 323
 - sys.modules, 375
 - słowniki, 24
 - filtrowanie, 28
 - kolejność elementów, 25

- obliczenia na danych, 26
- podzbiór słownika, 39
- redukcja danych, 27
- wspólne dane, 28
- słowo kluczowe
 - as, 506
 - compresslevel, 145
 - dir, 162
 - file, 139
 - metaclass, 325
 - namespace, 362
 - prefix, 162
 - property, 241
 - suffix, 162
- sortowanie
 - list słowników, 33
 - obiektów, 34
- specyfikator `__slots__`, 230
- sprawdzanie
 - sygnatury, 327
 - typów, 307
 - typu atrybutu, 232
 - występowania wyjątków, 501, 547
- standard
 - IEEE 754, 89
 - ISO 3166, 112
 - POSIX, 474
 - WSGI, 399–402
- statystyki, 200
- stoper, 493
- stosowanie
 - dekoratorów, 383
 - generatorów, 462
 - kontenerów, 521
 - niestandardowych separatorów, 140
- strefy czasowe, 111
- struktura, 192
 - cykliczna, 288
 - `Py_buffer`, 537
 - upraszczanie inicjowania, 248
- struktury danych, 15
- strumień
 - `stdout`, 497
 - `sys.stderr`, 476
 - `sys.stdout`, 473
 - tokenów, 73, 74
- sygnał SIGXCPU, 495
- sygnatura funkcji, 316, 327
- symbole wieloznaczne, 50
- synchronizowanie wątków, 434
- system
 - szesnastkowy, 187
 - typów, 254
 - sterowany zdarzeniami, 424

Ś

- ścieżka, 150, 366
- śląd błędu, 583

T

- tablica translacji, 63
- tablice, 535
- tablice bajtów, 83
- tekst
 - dopasowanie bloku, 57
 - dopasowanie najkrótsze, 56
 - dopasowywanie, 48, 50
 - formatowanie, 70
 - parsowanie, 62, 72, 75
 - postać standardowa, 58
 - wyrównywanie, 64
 - wyszukiwanie, 51, 54
 - zastępowanie, 54
 - zastępowanie encji, 71
- tekstowa reprezentacja obiektu, 225
- Telnet, 419, 424
- testowanie
 - danych wyjściowych, 497
 - kodu klienta HTTP, 392
 - serwera, 400
- testy jednostkowe, 498, 501
- tokeny, 73
- tryb pliku, 161
- tworzenie
 - archiwów, 484
 - atrybutów zarządzanych, 232
 - certyfikatu, 416
 - funkcji, 203, 204
 - generatora, 116
 - interfejsu, 399
 - iteratorów, 118
 - jednostki wywoływanej, 564
 - katalogów, 361
 - kolejki priorytetowej, 22
 - maszyny stanowej, 273
 - modelu danych, 254
 - modułów rozszerzeń, 553
 - nakładek, 297, 527, 550, 555
 - niestandardowych kontenerów, 259
 - niestandardowych wyjątków, 508
 - nowego rodzaju atrybutów, 243
- tworzenie
 - nowego środowiska, 385
 - obiektów, 267, 294, 534
 - obiektu metaklasy, 335
 - pakietów, 355
 - puli wątków, 446
 - rozszerzeń, 532

- tworzenie
 - serwera TCP, 393
 - serwera UDP, 395, 397
 - słownika, 24
 - stopera, 493
 - systemu typów, 254
 - wielosłownika, 37
 - współużytkowanej kolejki, 436
 - wydajnego kodu, 560, 562
 - zagnieżdżonych struktur, 280
 - zbioru, 30
- tymczasowe katalogi, 160
- typ atrybutu, 232

U

- udostępnianie certyfikatów, 415
- ułamki, 98
- Unicode, 58, 60
- unikanie
 - abstrakcji, 521
 - powtórzeń, 342
 - tworzenia kopii, 522
 - tworzenia struktur, 522
- uruchamianie
 - procesu demona, 471
 - przeglądarki internetowej, 495
- usługi HTTP, 389
- usuwanie
 - elementów, 20
 - powtórzeń, 29
 - węzłów, 179
 - znaków, 61
- utrata
 - cyfr znaczących, 90
 - metadanych, 299
- uwierzytelnianie, 368
- uwierzytelnianie klientów, 410

W

- wartość, 27
 - NaN, 96
 - Node, 287
 - None, 208
 - NULL, 547, 565
- wątki
 - blokowanie, 439, 441
 - obsługa synchronizacji, 440
 - problem uczujących filozofów, 444
 - synchronizowane, 433
 - tworzenie puli, 446
 - uruchamianie, 429
 - uruchamianie niedeterministyczne, 432
 - uruchamianie niezależnie, 432

- w językach C i Python, 549
- współużytkowana kolejka, 436
- wyścig, 439
- wzajemna komunikacja, 434
- zakleszczenie, 438, 442
- zamykanie, 430
- zapisywanie stanu, 445
- zatrzymywanie, 429
- wczytywanie
 - danych binarnych, 146, 158
 - danych CSV, 167
 - danych JSON, 170
 - dokumentu XML, 181
 - modułów, 368
 - pakietów, 372
 - plików konfiguracyjnych, 486
 - pliku z pakietu, 365
 - ponowne modułów, 362
 - zagnieżdżonych struktur, 192
- widoki pamięci, 562
- wielosłownik, 24
- wiersz poleceń, 477
- właściwości, 240
- wskaźnik, 529
 - do struktury danych, 538
 - do funkcji, 543
- wspólna nazwa pakietu, 361
- współbieżność, 429
- wycinek, 30
- wycinki danych, 121
- wydajne przetwarzanie tablic, 560
- wyjątek, 501
 - ActorExit, 457
 - DifferentException, 511
 - FileNotFoundError, 506
 - GeneratorExit, 467
 - ImportError, 379
 - MemoryError, 495
 - NotImplementedError, 275
 - OSError, 507
 - PermissionError, 506
 - RuntimeError, 509
 - SomeException, 511
 - StopIteration, 113, 117
 - SystemExit, 476
 - too many values to unpack, 16
 - UnicodeEncodeError, 154
 - ValueError, 169, 501
- wyjątki
 - hierarchie dziedziczenia, 506
 - klauzula except, 506
 - niestandardowe, 508
 - ponowne zgłaszanie, 512
 - przechwytywanie, 507

- wykonywanie
 - kodu z katalogu, 364
 - kodu z pliku zip, 364
 - poleceń w powłoce, 481
- wykorzystywanie pamięci, 494, 558
- wykrywanie zakleszczeń, 444
- wymiana potwierdzeń RTS-CTS, 163
- wyodrębnianie danych, 547
- wypakowywanie
 - elementów, 16
 - obiektów iterowalnych, 17
 - sekwencji, 15
- wrażenia regularne, 48, 52, 56, 61
- wyrażenie
 - lambda, 35, 136, 210
 - z gwiazdką, 17
- wyrównywanie łańcuchów, 64
- wysyłanie tablic, 427
- wyszukiwanie
 - identycznych danych, 28
 - największych elementów, 20
 - plików, 485
 - tekstu, 54, 55
 - wzorców tekstowych, 51
- wyświetlanie
 - komunikatów, 513
 - liczb, 90
 - śladu błędu, 583
 - nazwy pliku, 154
- wywołanie
 - basicConfig(), 490
 - RPC, 407, 409
 - swig, 552
 - yield, 132, 179
- wywołania
 - metod, 236, 278
 - poleceń zewnętrznych, 481
 - zwrotne, 221
- wzorzec
 - iterowania, 115
 - singleton, 321
 - tekstowy, 48, 56
 - z wyrażeniem regularnym, 48, 52

X

XML, 174

Z

- zachowywanie ostatnich elementów, 19
- zaokrąglanie liczb, 87
- zapis
 - danych binarnych, 141
 - danych CSV, 167

- danych JSON, 170
- danych tekstowych, 137
- danych wyjściowych testu, 503
- do pliku, 139, 142, 158
- dokumentu XML, 181
- plików skompresowanych, 144
- pliku tekstowego, 137
- stanu wątku, 445
- tablic binarnych, 188
- zarządzanie
 - blokadą GIL, 547
 - kontekstem, 228
 - nieprzejrzystymi wskaźnikami, 538
 - pamięcią, 288
 - sygnaturami funkcji, 318
 - systemem, 475
- zasady gramatyki, 77
- zastępowanie
 - encji, 72
 - klas, 215
 - tekstu, 54
 - znaków specjalnych, 71
- zastosowanie
 - dekoratorów, 562
 - kolejki, 22
 - metaklas, 322
- zbiór, 29
- zdalne wywołania procedur, 407
- zdarzenia, 422
- złożoność algorytmu, 522
- zmienianie kodowania, 569
- zmienna, 15
 - sys.argv, 479
 - sys.path, 366
- zmienna środowiskowa
 - PYTHONFAULTHANDLER, 582
 - PYTHONPATH, 366
- zmienne
 - lokalne, 520
 - z gwiazdką, 17
- zmniejszanie zużycia pamięci, 230
- znacznik końca wiersza, 54, 140
- znak gwiazdki, 17, 18
- znaki
 - podkreślenia, 231
 - znaki specjalne, 71
- zwalnianie blokady GIL, 548

Ż

- żądanie
 - GET, 389
 - HEAD, 391
 - POST, 391

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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 WYDAWNICZA

 **Helion SA**

Python. Receptury



Python to język programowania z ponad 20-letnią historią. Opracowany na początku lat 90. ubiegłego wieku, błyskawicznie zdobył sympatię programistów. Jest używany zarówno do pisania przydatnych skryptów czy małych narzędzi, jak i do pracy nad dużymi projektami. Korzysta z automatycznego zarządzania pamięcią oraz pozwala na podejście obiektowe i funkcyjne do tworzonego programu. Wokół języka Python skupiona jest bardzo silna społeczność programistów.

Ta książka to sprawdzone źródło informacji na temat Pythona i jego najczęstszych zastosowań. Należy ona do cenionej serii *Receptury*, w której znajdziesz najlepsze sposoby rozwiązywania problemów. Przekonaj się, jak wydajnie operować na strukturach danych, łańcuchach znaków, tekście i liczbach. Zobacz, jak korzystać z iteratorów i generatorów. Ponadto naucz się tworzyć własne klasy i funkcje oraz sprawdź, jak uzyskać dostęp do plików i sieci. Te i dziesiątki innych receptur opisano w tej książce. To obowiązkowa pozycja na półce każdego programisty pracującego z językiem Python.

Dzięki tej książce:

- poznasz podstawy języka Python
- w optymalny sposób rozwiążesz najczęstsze problemy
- napiszesz program korzystający z puli wątków
- będziesz lepszym programistą Pythona!

Najlepsze rozwiązania typowych problemów!

helion.pl
księgarnia
internetowa

Nr katalogowy: 17110



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:

📍 <http://helion.pl/promocje>

Książki najchętniej czytane:

📍 <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

📍 <http://helion.pl/nowosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

sięgnij po WIECEJ



KOD KORZYŚCI

ISBN 978-83-246-8180-8



Cena 99,00 zł