PROGRAMOWANIE W PYTHONIE DLA ŚREDNIO ZAAWANSOWANYCH

NAJLEPSZE PRAKTYKI TWORZENIA CZYSTEGO KODU

AL SWEIGART

Tytuł oryginału: Beyond the Basic Stuff with Python: Best Practices for Writing Clean Code

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-8338-8

Copyright © 2021 by Al Sweigart. Title of English-language original: Beyond the Basic Stuff with Python: Best Practices for Writing Great Code, ISBN 9781593279660 published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103. The Polish-language edition Copyright © 2022 by Helion S.A. under license by No Starch Press Inc. All rights reserved.

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

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

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

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

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 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/prpysz
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

WPROWADZENIE **13**

Część I. Pierwsze kroki 19

1
OBSŁUGA BŁĘDÓW I POSZUKIWANIE POMOCY **21**

Komunikaty o błędach w Pythonie 22
- Analiza śladu 22
- Wyszukiwanie informacji na temat komunikatów o błędach 25

Zapobieganie błędom dzięki wykorzystaniu linterów 26
Jak prosić o pomoc w programowaniu? 28
- Ogranicz konieczność dopowiadania — od razu podaj jak najwięcej informacji 29
- Sformułuj swój problem w postaci rzeczywistego pytania 29
- Zadawaj pytania na właściwej stronie internetowej 29
- Podsumuj swoje pytanie w nagłówku 30
- Wyjaśnij, co ma robić Twój kod 30
- Dołącz kompletny komunikat o błędzie 30
- Udostępnij swój pełny kod 30
- Zastosuj odpowiednie formatowanie, aby poprawić czytelność kodu 32
- Powiedz osobom pomagającym, czego już próbowałeś 33
- Opisz swoją konfigurację 33

Przykłady pytań 34
Podsumowanie 34

2
KONFIGURACJA ŚRODOWISKA I WIERSZ POLECENIA **36**

System plików 37
- Ścieżki w Pythonie 38
- Katalog domowy 38
- Bieżący katalog roboczy 39
- Ścieżki względne i bezwzględne 39

Programy i procesy 40
Wiersz poleceń 41
- Otwieranie okna terminala 42
- Uruchamianie programów z wiersza poleceń 43
- Korzystanie z argumentów wiersza poleceń 44

Uruchamianie kodu Pythona z wiersza poleceń z wykorzystaniem argumentu -c 46
Uruchamianie programów w Pythonie z wiersza poleceń 46
Uruchamianie programu py.exe 46
Uruchamianie poleceń systemu operacyjnego z programu Pythona 47
Minimalizacja wpisywania dzięki mechanizmowi uzupełniania z wykorzystaniem klawisza Tab 48
Wyświetlanie historii poleceń 48
Korzystanie z popularnych poleceń 49
Zmienne środowiskowe i PATH 57
Wyświetlanie wartości zmiennych środowiskowych 57
Korzystanie ze zmiennej środowiskowej PATH 58
Zmiana zawartości zmiennej środowiskowej PATH dla wiersza polecenia 59
Trwałe dodawanie folderów do zmiennej PATH w systemie Windows 60
Trwałe dodawanie folderów do zmiennej PATH w systemach macOS i Linux 61
Uruchamianie programów Pythona bez wiersza poleceń 61
Uruchamianie programów Pythona w systemie Windows 62
Uruchamianie programów Pythona w systemie macOS 63
Uruchamianie programów Pythona w systemie Ubuntu Linux 63
Podsumowanie 64

Część II. Najlepsze praktyki, narzędzia i techniki 65

3
FORMATOWANIE KODU ZA POMOCĄ NARZĘDZIA BLACK 67
Jak stracić przyjaciół i zrobić sobie wrogów wśród współpracowników? 68
Przewodniki stylu i PEP 8 68
Odstępy w poziomie 69
Używaj jako wcięć znaków spacji 69
Odstępy w obrębie wiersza 71
Odstępy w pionie 74
Przykład zastosowania odstępów w pionie 75
Najlepsze praktyki dotyczące odstępów w pionie 76
Black: bezkompromisowy formater kodu 77
Instalacja narzędzia Black 77
Uruchamianie narzędzia Black z wiersza polecenia 78
Wyłączenie programu Black dla części Twojego kodu 81
Podsumowanie 82

4
WYBIERANIE ZROZUMIAŁYCH NAZW 83
Style wielkości liter 84
Konwencje nazewnictwa PEP 8 85
Odpowiednia długość nazw 85
Zbyt krótkie nazwy 86
Zbyt długie nazwy 87
Korzystaj z nazw ułatwiających wyszukiwanie 89
Unikaj dowcipów, kalamburów i określeń żargonowych 89
Nie nadpisuj wbudowanych nazw 90
Najgorsze możliwe nazwy zmiennych 91
Podsumowanie 92

Kup ksi k Pole ksi k

5
WYSZUKIWANIE CUCHNĄCEGO KODU 93
Powielony kod 94
Magiczne liczby 95
Kod wykomentowany i martwy 98
Debugowanie za pomocą komunikatów 100
Zmienne z przyrostkami numerycznymi 101
Klasy, które powinny być funkcjami lub modułami 101
Listy składane wewnątrz list składanych 102
Puste bloki except i niejasne komunikaty o błędach 104
Mity związane z cuchnącym kodem 106
Mit: funkcje powinny mieć tylko jedną instrukcję return na końcu 106
Mit: funkcje powinny zawierać co najwyżej jedną instrukcję try 106
Mit: argumenty-flagi są złe 107
Mit: zmienne globalne są złe 108
Mit: komentarze są niepotrzebne 109
Podsumowanie 110

6
PISANIE „PYTHONICZNEGO” KODU 111
Zen Pythona 112
Naucz się cenić znaczące wcięcia 115
Częste przypadki niewłaściwego korzystania ze składni 116
Używaj funkcji enumerate() zamiast range() 117
Używaj instrukcji with zamiast open() i close() 118
Do porównywania z None używaj is zamiast == 119
Formatowanie ciągów znaków 119
Jeśli ciąg zawiera wiele lewych ukośników, używaj surowych ciągów znaków 120
Formatowanie ciągów za pomocą f-stringów 120
Tworzenie płytkich kopii list 122
Pythoniczne sposoby korzystania ze słowników 123
Używaj ze słownikami wywołań get() i setdefault() 123
Użyj dla wartości domyślnych klasy collections.defaultdict 124
Używaj słowników zamiast instrukcji switch 125
Wyrażenia warunkowe: „brzydki” operator trójargumentowy Pythona 126
Korzystanie z wartości zmiennych 128
Operatory przypisania i operatory porównania 128
Sprawdzanie, czy zmienna jest jedną z wielu wartości 129
Podsumowanie 130

7
PROGRAMISTYCZNY ŻARGON 131
Definicje 132
Python język i Python interpreter 132
Odśmiecanie 133
Literały 133
Słowa kluczowe 134
Obiekty, wartości, egzemplarze i tożsamości 135
Elementy 138
Obiekty mutowalne i niemutowalne 138
Indeksy, klucze i skróty 141

Kup ksi k Pole ksi k

Kontenery, sekwencje, mapowanie i typy zbiorów 144
Metody dunder i metody magiczne 145
Moduły i pakiety 145
Obiekty wywoływalne i obiekty pierwszej klasy 146
Często mylone terminy 147
Instrukcje a wyrażenia 147
Blok, klauzula i ciało 148
Zmienna a atrybut 149
Funkcja a metoda 150
Obiekt iterowalny a iterator 150
Błędy składniowe, błędy wykonania a błędy semantyczne 152
Parametry a argumenty 153
Koercja typu a rzutowanie typu 154
Właściwości a atrybuty 154
Kod bajtowy a kod maszynowy 155
Skrypt a program, język skryptowy a język programowania 155
Biblioteka, framework, SDK, silnik i API 156
Podsumowanie 157
Dalsza lektura 158

8

ZNANE PUŁAPKI PYTHONA 159

Nie dodawaj ani nie usuwaj elementów z listy, kiedy po niej iterujesz 160
Nie kopiuj mutowalnych wartości inaczej niż poprzez wywołania copy.copy()
lub copy.deepcopy() 166
Nie używaj wartości mutowalnych w roli argumentów domyślnych 169
Nie buduj ciągów za pomocą konkatenacji 171
Nie oczekuj, że funkcja sort() posortuje listę alfabetycznie 173
Nie zakładaj, że liczby zmiennoprzecinkowe są idealnie dokładne 174
Nie twórz łańcucha operatorów nierówności != 177
Nie zapominaj o przecinku w krotce złożonej z jednego elementu 178
Podsumowanie 178

9

EZOTERYCZNE OSOBLIWOŚCI PYTHONA 180

Dlaczego 256 to jest 256, ale 257 to nie jest 257 180
Internowanie ciągów 182
Sztuczne operatory inkrementacji i dekrementacji w Pythonie 183
Wszystko z nic 185
Wartości logiczne są liczbami całkowitymi 186
Tworzenie łańcucha operatorów różnego rodzaju 187
Antygrawitacja w Pythonie 188
Podsumowanie 189

10

PISANIE SKUTECZNYCH FUNKCJI 190

Nazwy funkcji 190
Kompromisy dotyczące rozmiaru funkcji 191
Parametry i argumenty funkcji 194
Argumenty domyślne 194
Korzystanie z * i ** w celu przekazywania argumentów do funkcji 195

Kup ksi k Pole ksi k

Użycie * do tworzenia funkcji wariadycznych 197
Tworzenie funkcji wariadycznych za pomocą składni ** 199
Tworzenie funkcji-wrapperów za pomocą składni * i ** 201
Programowanie funkcyjne 202
Skutki uboczne 202
Funkcje wyższego rzędu 204
Funkcje lambda 205
Mapowanie i filtrowanie z wykorzystaniem list składanych 206
Zwracane wartości zawsze powinny mieć ten sam typ danych 207
Zgłaszanie wyjątków a zwracanie kodów błędów 209
Podsumowanie 210

11
KOMENTARZE, DOCSTRINGI I WSKAZÓWKI TYPU 211
Komentarze 212
Styl komentarzy 213
Komentarze inline 214
Komentarze wyjaśniające 214
Komentarze podsumowujące 215
Komentarze typu „wyciągnięte wnioski” 216
Komentarze prawne 216
Profesjonalny ton 217
Znaczniki kodu i komentarze TODO 217
Magiczne komentarze i kodowanie plików źródłowych 218
Docstringi 218
Wskazówki typu 221
Korzystanie z narzędzi do statycznej analizy kodu 223
Ustawianie wskazówek dla wielu typów 225
Ustawianie wskazówek typu dla list, słowników i nie tylko 226
Backport wskazówek typu z wykorzystaniem komentarzy 227
Podsumowanie 229

12
ORGANIZOWANIE PROJEKTÓW KODU Z WYKORZYSTANIEM SYSTEMU GIT 230
Commity i repozytoria systemu Git 231
Korzystanie z narzędzia Cookiecutter do tworzenia nowych projektów w Pythonie 231
Instalacja Gita 234
Konfigurowanie nazwy użytkownika Gita i adresu e-mail 234
Instalacja narzędzi GUI dla Gita 235
Przepływ pracy w systemie Git 236
W jaki sposób Git śledzi stan pliku? 236
Po co jest stan staged? 238
Tworzenie repozytorium Gita na komputerze lokalnym 238
Dodawanie plików do śledzenia przez Gita 240
Ignorowanie plików w repozytorium 241
Zatwierdzanie zmian 242
Usuwanie plików z repozytorium 247
Zmiana nazwy i przenoszenie plików w repozytorium 248
Przeglądanie loga commitów 249
Przywracanie wcześniejszych zmian 250
Cofanie niezatwierdzonych lokalnych zmian 251
Anulowanie stanu staged pliku 251

Kup ksi k Pole ksi k

Cofanie najnowszych commitów 252
Cofanie zmian do określonego commita dla pojedynczego pliku 252
Przepisywanie historii commitów 254
GitHub i polecenie git push 254
Przesyłanie istniejącego repozytorium do usługi GitHub 255
Klonowanie repozytorium z istniejącego repozytorium w serwisie GitHub 256
Podsumowanie 257

13
MIERZENIE WYDAJNOŚCI ALGORYTMÓW I ANALIZA BIG O 258
Moduł timeit 259
Profiler cProfile 261
Analiza algorytmów Big O 263
Rzędy w notacji Big O 264
Metafora regału dla rzędów notacji Big O 265
Notacja Big O mierzy najgorszy scenariusz 269
Określanie rzędu Big O kodu 271
Dlaczego niższe rzędy i współczynniki nie mają znaczenia? 272
Przykłady analizy Big O 273
Rzędy Big O popularnych wywołań 276
Analiza Big O w mgnieniu oka 277
Big O nie ma znaczenia dla małych n, czyli dla większości przypadków 279
Podsumowanie 279

14
PRAKTYCZNE PROJEKTY 281
Wieża Hanoi 282
Wyjście 283
Kod źródłowy 284
Pisanie kodu 286
Cztery w rzędzie 294
Wyjście 295
Kod źródłowy 296
Pisanie kodu 299
Podsumowanie 308

Część III. Python obiektowy 309

15
KLASY I PROGRAMOWANIE OBIEKTOWE 311
Analogia do rzeczywistego świata: wypełnianie formularza 312
Tworzenie obiektów na podstawie klas 314
Tworzenie prostej klasy: WizCoin 315
Metody __init__() i self 317
Atrybuty 318
Atrybuty prywatne i metody prywatne 319
Funkcja type() i atrybut __qualname__ 321
Przykłady kodu obiektowego i nieobiektowego: kółko i krzyżyk 322
Projektowanie klas dla rzeczywistych aplikacji jest trudne 327
Podsumowanie 328

Kup ksi k Pole ksi k

16
PROGRAMOWANIE OBIEKTOWE I DZIEDZICZENIE ...329

Jak działa dziedziczenie 330
Przesłanianie metod 332
Funkcja super() 334
Staraj się stosować kompozycję zamiast dziedziczenia 336
Minusy dziedziczenia 337
Funkcje isinstance() i issubclass() 340
Metody klasy 341
Atrybuty klasy 343
Metody statyczne 344
Kiedy używać metod i atrybutów klasy oraz metod statycznych w programach obiektowych? 344
Terminologia obiektowa 345
Hermetyzacja 345
Polimorfizm 345
Kiedy nie używać dziedziczenia? 346
Dziedziczenie wielokrotne 346
Kolejność rozpoznawania metod 348
Podsumowanie 350

17
PYTHONICZNY PARADYGMAT OOP: WŁAŚCIWOŚCI I METODY DUNDER ...352

Właściwości 353
Przekształcanie atrybutu we właściwość 353
Używanie setterów do sprawdzania poprawności danych 356
Właściwości tylko do odczytu 358
Kiedy używać właściwości? 359
Metody dunder w Pythonie 360
Metody dunder reprezentacji tekstowych 360
Liczbowe metody dunder 363
Odbite numeryczne metody dunder 366
Metody dunder rozszerzonego przypisania w miejscu 369
Metody dunder porównań 370
Podsumowanie 376

Kup ksi k
Pole ksi k

Kup ksi k

Pole ksi k

6

Pisanie „pythonicznego” kodu

POTĘŻNY (ANG. *POWERFUL*) TO BEZSENSOWNY PRZYMIOTNIK UŻYWANY DO OPISYWANIA JĘZYKÓW PROGRAMOWANIA. TWÓRCY KAŻDEGO JĘZYKA PROGRAMOWANIA TWIERDZĄ, ŻE ICH JĘZYK JEST POTĘŻNY. Oficjalny podręcznik Pythona zaczyna się od zdania „Python jest łatwym do nauczenia się, potężnym językiem programowania”. Nie istnieje jednak algorytm pozwalający obliczyć, co w jednym języku można zrobić, a czego w innym nie można, i nie ma jednostki miary pozwalającej zmierzyć ilościowo „moc” języka programowania (choć z pewnością można zmierzyć siłę głosu, z jaką programiści twierdzą, że ich ulubiony język jest lepszy od innych).

Każdy język charakteryzuje się własnymi wzorcami projektowymi i zawiera haczyki, które składają się na jego mocne i słabe strony. Aby napisać kod w Pythonie jak prawdziwy pythonista, musisz wiedzieć znacznie więcej, nie tylko znać składnię i standardową bibliotekę. Następnym krokiem jest poznanie *idiomów*, czyli specyficznych dla Pythona praktyk kodowania. Niektóre własności języka Python ułatwiają pisanie kodu w sposób, który określa się jako *pythoniczny*.

W tym rozdziale przedstawię kilka często stosowanych sposobów pisania idiomatycznego kodu w Pythonie wraz z ich niepythonicznymi odpowiednikami. To, co uznajemy za styl pythoniczny, może być różnie postrzegane przez programistów, ale często składają się na niego przykłady i praktyki, które tutaj omawiam. Opisanych technik używają doświadczeni programiści Pythona, więc zapoznanie się z nimi pozwala rozpoznać je w rzeczywistym kodzie.

Kup ksi k   Pole ksi k

Zen Pythona

Zen Pythona to opracowany przez Tima Petersa zbiór 20 wytycznych dotyczących projektowania w języku Python oraz pisania programów w Pythonie. Twój kod w Pythonie nie musi być zgodny z tymi wytycznymi, ale warto o nich pamiętać. Zen Pythona wyświetla się również jako tzw. „wielkanocne jajo”, czyli ukryty żart, który pojawia się po uruchomieniu polecenia `import this`:

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
--ciach--
```

UWAGA *Co ciekawe, zapisano tylko 19 wytycznych. Guido van Rossum, twórca Pythona, podobno powiedział, że brakujący 20. aforyzm to „jakiś dziwaczny żart Tima Petersa”. Tim pozostawił go Guidowi, by ten go uzupełnił, ale to nigdy nie nastąpiło.*

Ostatecznie wytyczne te są opiniami, z którymi programiści mogą się zgadzać lub nie. Podobnie jak wszystkie dobre zbiory zasad moralnych, bywają one sprzeczne ze sobą, co pozwala zapewnić największą elastyczność. Oto moja interpretacja tych aforyzmów:

Beautiful is better than ugly (*piękne jest lepsze niż brzydkie*). Piękny kod to kod czytelny i zrozumiały. Programiści często piszą kod szybko i nie zwracają uwagi na jego czytelność. Komputer bez trudu potrafi uruchomić nieczytelny kod, ale nieczytelny kod jest trudny do utrzymania i debugowania przez programistów-ludzi. Piękno jest subiektywne, ale kod napisany w sposób, który nie bierze pod uwagę aspektu czytelności, jest często uznawany za brzydki. Powodem popularności Pythona jest fakt, że jego składnia, w odróżnieniu od innych języków, nie jest zaśmiecona tajemniczymi znakami interpunkcyjnymi, co znacznie ułatwia pracę.

Explicit is better than implicit (*jawne jest lepsze niż niejawne*). Gdybym napisał tylko „to jest oczywiste”, byłoby to nic niemówiące wyjaśnienie tego aforyzmu. Podobnie jest w kodzie. Należy dążyć do wyrażania się w sposób pełny i jawny. Należy unikać ukrywania funkcjonalności kodu za niejasnymi własnościami języka, ponieważ zrozumienie takiego kodu wymaga głębokiej znajomości wielu szczegółów.

Simple is better than complex (*proste jest lepsze niż złożone*). **Complex is better than complicated** (*złożone jest lepsze niż skomplikowane*). Te dwa aforyzmy przypominają, że każdy kod możemy zbudować za pomocą technik prostych lub złożonych. Jeśli masz do wykonania prostą pracę, do której wystarczy łopata, to użycie 50-tonowego buldożera hydraulicznego jest przesadą. Jednak w przypadku większej pracy złożoność obsługi jednego buldożera jest lepsza niż komplikacje związane z koordynacją zespołu stu łopat. Preferuj prostotę zamiast złożoności, ale znaj granice prostoty.

Kup ksi k Pole ksi k

Flat is better than nested (*płaskie jest lepsze niż zagnieżdżone*). Programiści uwielbiają organizować swój kod w kategorie. Te kategorie zawierają podkategorie zawierające inne podkategorie. Takie hierarchie często w większym stopniu wprowadzają biurokrację niż organizację. Nie ma niczego złego w pisaniu kodu składającego się tylko z jednego modułu najwyższego poziomu lub z jednej struktury danych. Jeśli w Twoim kodzie są instrukcje w postaci `spam.eggs.bacon.ham()` lub `spam['eggs']['bacon']['ham']`, to prawdopodobnie jest on zbyt skomplikowany.

Sparse is better than dense (*rzadkie jest lepsze niż gęste*). Programiści często lubią wciskać jak najwięcej funkcjonalności do jak najmniejszej ilości kodu, tak jak w poniższym wierszu: `print('\n'.join("%i bajty (ów) = %i bitów, czyli %i możliwych wartości." % (j, j*8, 256**j-1) for j in (1 << i for i in range(8))))`. Chociaż taki kod może zaimponować Twoim znajomym, może również doprowadzać do pasji Twoich współpracowników, którzy będą starali się go zrozumieć. Nie staraj się tworzyć kodu „zbyt gęstego", czyli takiego, który robi zbyt wiele naraz. Kod rozłożony na wiele wierszy jest często łatwiejszy do czytania niż gęste jednolinijkowce. Ten aforyzm jest podobny do tego, który mówi, że proste jest lepsze niż skomplikowane.

Readability counts (*czytelność się liczy*). Chociaż nazwa `strcmp()` w znaczeniu „porównaj ciągi" może być oczywista dla kogoś, kto programuje w C od lat 70., to nowoczesne komputery mają wystarczająco dużo pamięci, aby można było zapisać pełną nazwę funkcji. Nie pomijaj liter z Twoich nazw ani nie pisz nadmiernie zwięzłego kodu. Poświęć trochę czasu, aby wymyślić opisowe, konkretne nazwy dla zmiennych i funkcji. Pusty wiersz pomiędzy fragmentami kodu może pełnić tę samą funkcję co podziały akapitów w książce — informować czytelnika o tym, które części powinny być czytane razem. Ten aforyzm jest podobny do tego, który mówi, że „piękne jest lepsze niż brzydkie".

Special cases aren't special enough to break the rules. Although practicality beats purity. (*specjalne przypadki nie są wystarczająco specjalne, aby usprawiedliwiały łamanie zasad; względy praktyczne powinny jednak brać górę nad względami czystości przestrzegania zasad*). Te dwa aforyzmy są ze sobą sprzeczne. Programowanie jest pełne „najlepszych praktyk", do których stosowania programiści powinni dążyć w swoim kodzie. Naruszenie tych praktyk po to, by szybko napisać krótki program, może być kuszące, ale może prowadzić do stworzenia „szczurzego gniazda" niespójnego i nieczytelnego kodu. Z drugiej strony dążenie za wszelką cenę do przestrzegania reguł może spowodować powstanie nadmiernie abstrakcyjnego i nieczytelnego kodu. Na przykład dążenie w kodzie Javy do tego, by cały kod był dopasowany do paradygmatu obiektowego, często powoduje konieczność tworzenia dużej ilości kodu typu *boilerplate* w celu stworzenia nawet najmniejszego programu. Znalezienie równowagi pomiędzy tymi dwoma aforyzmami staje się łatwiejsze wraz ze zdobywanym doświadczeniem. Z czasem nie tylko

Kup ksi k Pole ksi k

poznasz zasady, ale także nauczysz się rozpoznawać sytuacje, kiedy ich łamanie jest usprawiedliwione.

Errors should never pass silently. Unless explicitly silenced (*błędy nigdy nie powinny przechodzić bez echa, chyba że je specjalnie wyciszyłeś*). To, że programiści często ignorują komunikaty o błędach, nie oznacza bynajmniej, że program powinien przestać je emitować. „Ciche" błędy mogą się zdarzyć, gdy zamiast wywoływać wyjątki, funkcje zwracają kody błędów lub `None`. Te dwa aforyzmy mówią nam, że lepiej, aby program szybko zawiódł i uległ awarii, niż by błąd został wyciszony i program kontynuował działanie. Błędy, które nieuchronnie zdarzą się później, będą trudniejsze do debugowania, ponieważ zostaną wykryte długo po wystąpieniu pierwotnej przyczyny. Chociaż zawsze możesz zdecydować się na jawne ignorowanie błędów powstających w programach, dokonuj świadomego wyboru.

In the face of ambiguity, refuse the temptation to guess (*jeśli natkniesz się na dwuznaczność, nie ulegaj pokusie zgadywania*). Komputery uczyniły ludzi przesądnymi: w procesie „odprawiania egzorcyzmów" nad demonami w naszych komputerach wykonujemy święty rytuał wyłączania komputera, a następnie włączania go ponownie. Podobno to rozwiązuje każdy tajemniczy problem. Komputery nie są jednak magią. Jeśli kod nie działa, istnieje powód, dla którego nie działa, i tylko ostrożne, krytyczne myślenie może doprowadzić do rozwiązania problemu. Nie ulegaj pokusie ślepego wypróbowywania rozwiązań, aż coś zacznie działać; często w ten sposób tylko zamaskujesz problemu, a go nie rozwiążesz.

There should be one — and preferably only one — obvious way to do it (*powinien istnieć co najmniej jeden, a najlepiej tylko jeden, oczywisty sposób, aby to zrobić*). Jest to manifest przeciwko regule wyrażonej w motcie języka programowania Perl: „Istnieje więcej niż jeden sposób, aby to zrobić!". Okazuje się, że stosowanie trzech lub czterech różnych sposobów pisania kodu, który wykonuje to samo zadanie, jest mieczem obosiecznym: zyskujesz elastyczność w sposobie pisania kodu, ale aby odczytać kod napisany przez innych, musisz poznać wszystkie możliwe sposoby, w jaki mógł zostać on napisany. Elastyczność nie jest warta zwiększonego wysiłku potrzebnego do nauczenia się języka programowania.

Although that way may not be obvious at first unless you're Dutch (*ten sposób może jednak nie być natychmiast oczywisty, chyba że jesteś Holendrem*). Ten aforyzm to żart. Guido van Rossum, twórca Pythona, jest Holendrem.

Now is better than never. Although never is often better than *right* now (*teraz to lepiej niż nigdy; chociaż nigdy często jest lepsze niż natychmiast*). Te dwa aforyzmy mówią nam, że kod, który działa powoli, jest w oczywisty sposób gorszy niż kod, który działa szybko. Lepiej jest jednak czekać na zakończenie programu, niż zakończyć go zbyt wcześnie z nieprawidłowymi wynikami.

Kup ksi k Pole ksi k

If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea (*jeśli implementacja jest trudna do wytłumaczenia, jest zła; jeśli implementacja jest łatwa do wyjaśnienia, może być dobra*). Z czasem wiele rzeczy się komplikuje: przepisy podatkowe, romantyczne związki, książki dotyczące programowania w Pythonie. Nie inaczej jest w przypadku oprogramowania. Te dwa aforyzmy przypominają nam, że jeśli kod jest tak skomplikowany, że programistom trudno jest go zrozumieć i debugować, to jest złym kodem. Jednak to, że kod programu łatwo jest wyjaśnić komuś innemu, nie oznacza, że nie jest to zły kod. Niestety zdecydowanie o tym, jak sprawić, by kod był tak prosty, jak to możliwe, ale nie prostszy, jest trudne.

Namespaces are one honking great idea — let's do more of those! (*przestrzenie nazw to świetny pomysł, korzystaj z nich jak najczęściej!*). Przestrzenie nazw to odrębne kontenery dla identyfikatorów, pozwalające zapobiec konfliktom nazewnictwa. Na przykład wbudowana funkcja `open()` i funkcja `webbrowser.open()` mają taką samą nazwę, ale odnoszą się do różnych funkcji. Zaimportowanie modułu `webbrowser` nie zastępuje wbudowanej funkcji `open()`, ponieważ obie funkcje `open()` istnieją w różnych przestrzeniach nazw: odpowiednio we wbudowanej przestrzeni nazw i przestrzeni nazw modułu `webbrowser`. Należy jednak zapamiętać, że płaskie jest lepsze niż zagnieżdżone: chociaż przestrzenie nazw są doskonałe, należy je tworzyć tylko po to, aby zapobiec konfliktom nazewnictwa, a nie po to, by wprowadzać niepotrzebne kategorie.

Podobnie jak w przypadku wszystkich opinii na temat programowania, można spierać się z tymi regułami, które wymieniłem powyżej. W Twojej sytuacji mogą one być po prostu bez znaczenia. Debatowanie o tym, jak należy pisać kod lub co zalicza się do kodu „pythonicznego", rzadko jest tak produktywne, jak myślisz (chyba że piszesz książkę pełną opinii dotyczących programowania).

Naucz się cenić znaczące wcięcia

Wśród programistów kodujących w innych językach największy niepokój w Pythonie budzą *znaczące wcięcia* w tym języku (często błędnie nazywane *znaczącymi odstępami*). Uznają je za dziwne i nieznane. Liczba wcięć na początku wiersza kodu ma znaczenie w Pythonie, ponieważ określa, które wiersze kodu znajdują się w tym samym bloku.

Grupowanie bloków kodu w Pythonie przy użyciu wcięć może wydawać się dziwne, ponieważ inne języki rozpoczynają i kończą swoje bloki za pomocą nawiasów klamrowych { i }. Jednak programiści kodujący w językach innych niż Python zwykle stosują wcięcia bloków kodu, podobnie jak programiści Pythona, aby ich kod stał się bardziej czytelny. Na przykład w języku programowania Java nie ma znaczących wcięć. Programiści Javy nie muszą stosować wcięć dla bloków kodu, ale często to robią, aby poprawić czytelność swoich programów.

Kup ksi k Pole ksi k

Poniższy przykład to funkcja Javy o nazwie `main()`, która zawiera pojedyncze wywołanie funkcji `println()`:

```
// Przykład z Javy
public static void main(String[] args) {
    System.out.println("Witaj, świecie!");
}
```

Ten kod w Javie działałby dobrze także wtedy, gdyby wiersz z wywołaniem funkcji `println()` nie został wcięty. To dlatego, że w Javie to nawiasy klamrowe, a nie wcięcia oznaczają początki i zakończenia bloków. Zamiast zezwalać na to, by wcięcia były opcjonalne, Python zmusza Cię, by kod był konsekwentnie czytelny. Należy jednak pamiętać, że w Pythonie nie ma *znaczących odstępów*, ponieważ nie istnieje ograniczenie sposobu używania odstępów nietworzących wcięć (zarówno `2 + 2`, jak i `2+2` są poprawnymi wyrażeniami języka Python).

Niektórzy programiści twierdzą, że klamra otwierająca powinna znajdować się w tym samym wierszu co instrukcja otwierająca, podczas gdy inni są zdania, że powinna znajdować się w następnym wierszu. Programiści zawsze będą głosić zalety preferowanego przez siebie stylu. Python starannie pomija ten problem dzięki całkowitej rezygnacji z nawiasów klamrowych. Dzięki temu programista Pythona może się skupić na bardziej produktywnej pracy. Osobiście chciałbym, aby we wszystkich językach programowania zostało przyjęte podejście Pythona do grupowania bloków kodu.

Niektórzy programiści nadal jednak tęsknią za nawiasami klamrowymi i chcieliby dodać je do przyszłych wersji Pythona — niezależnie od tego, jak bardzo są bez sensu. Moduł `__future__` pozwala na przenoszenie nowości do wcześniejszych wersji Pythona. Jeśli spróbujesz zaimportować moduł `braces` do Pythona, napotkasz na „wielkanocne jajo":

```
>>> from __future__ import braces
SyntaxError: not a chance
```

Nie liczyłbym na dodanie w najbliższym czasie nawiasów klamrowych do Pythona.

Częste przypadki niewłaściwego korzystania ze składni

Jeśli Python nie jest Twoim pierwszym językiem programowania, możesz pisać kod w nim, stosując takie same strategie, jakich używałeś do pisania kodu w innych językach programowania. A może nauczyłeś się niezwykłego sposobu pisania kodu w Pythonie, ponieważ nie wiedziałeś o istnieniu bardziej ugruntowanych, najlepszych rozwiązań. Taki dziwny kod działa, ale mógłbyś zaoszczędzić

Kup ksi k Pole ksi k

trochę czasu i wysiłku, gdybyś nauczył się bardziej standardowych sposobów pisania pythonicznego kodu. W tym podrozdziale opisałem typowe błędy popełniane przez programistów oraz wymieniłem sposoby, jak należy pisać kod prawidłowo.

Używaj funkcji enumerate() zamiast range()

Podczas przetwarzania w pętli listy lub innej sekwencji niektórzy programiści, w celu generowania liczb całkowitych oznaczających indeks od 0 do długości sekwencji (bez niej), stosują funkcje `range()` i `len()`. Często w tych pętlach używają nazwy zmiennej `i` (od słowa *index*). Oto przykład niephytonicznego kodu. Wprowadź go do interaktywnej powłoki, aby go wypróbować:

```
>>> animals = ['kot', 'pies', 'łoś ']
>>> for i in range(len(animals)):
...     print(i, animals[i])
...
0 kot
1 pies
2 łoś
```

Konwencja `range(len())` jest prosta, ale daleka od ideału, ponieważ może być nieczytelna. Zamiast tego przekaż listę lub sekwencję do wbudowanej funkcji `enumerate()`, która zwraca całkowitą wartość indeksu oraz element spod tego indeksu. Na przykład możesz napisać następujący pythoniczny kod:

```
>>> # Przykład kodu pythonicznego
>>> animals = ['kot', 'pies', 'łoś']
>>> for i, animal in enumerate(animals):
...     print(i, animal)
...
0 kot
1 pies
2 łoś
```

Jeśli użyjesz funkcji `enumerate()` zamiast `range(len())`, Twój kod stanie się nieco czystszy. Jeśli potrzebujesz samych elementów, a nie są Ci potrzebne indeksy, nadal możesz bezpośrednio iterować po liście w sposób pythoniczny:

```
>>> animals = ['kot', 'pies', 'łoś ']
>>> for animal in animals:
...     print(animal)
...
kot
pies
łoś
```

Wywoływanie funkcji `enumerate()` i bezpośrednie iterowanie po sekwencji jest lepsze niż użycie staromodnej konwencji `range(len())`.

Używaj instrukcji with zamiast open() i close()

Funkcja `open()` zwraca obiekt reprezentujący plik zawierający metody do odczytywania lub zapisywania pliku. Po zakończeniu wykonywania działań na pliku metoda `close()` obiektu reprezentującego plik udostępnia go do odczytu i zapisu innym programom. Wymienione funkcje można wykorzystywać indywidualnie. Postępowanie w ten sposób jest jednak niepythoniczne. Na przykład aby zapisać tekst „Witaj, świecie!" do pliku o nazwie *spam.txt*, wprowadź w interaktywnej powłoce następujące instrukcje:

```
>>> # Przykład niepythoniczny
>>> fileObj = open('spam.txt', 'w')
>>> fileObj.write('Witaj, świecie!')
13
>>> fileObj.close()
```

Pisanie kodu w taki sposób może prowadzić do sytuacji, w której plik pozostanie niezamknięty, jeśli na przykład wystąpi błąd w bloku `try` i program pominie wywołanie funkcji `close()`. Przykładowo:

```
>>> # Przykład niepythoniczny
>>> try:
...     fileObj = open('spam.txt', 'w')
...     eggs = 42 / 0 # Tutaj występuje dzielenie przez zero.
...     fileObj.close() # Ten wiersz nigdy się nie uruchomi.
... except:
...     print('Wystąpił błąd.')
...
Wystąpił błąd.
```

Po wystąpieniu błędu dzielenia przez zero sterowanie przechodzi do bloku `except`, z pominięciem wywołania metody `close()`, i pozostawia plik otwarty. Może to później prowadzić do błędów uszkodzenia pliku, które trudno przypisać instrukcjom w bloku `try`.

Zamiast sekwencji `open()`...`close()` możesz użyć instrukcji `with`, aby automatycznie wywołać metodę `close()`, w chwili gdy sterowanie opuszcza blok instrukcji `with`. W poniższym przykładzie pythonicznego kodu wykonywane jest to samo zadanie co w kodzie z pierwszego przykładu w tym punkcie:

```
>>> # Przykład pythoniczny
>>> with open('spam.txt', 'w') as fileObj:
...     fileObj.write('Witaj, świecie!')
...
```

Mimo że nie ma jawnego wywołania funkcji `close()`, instrukcja `with` „będzie wiedziała", żeby ją wywołać, gdy sterowanie opuszcza blok.

Kup ksi k Pole ksi k

Do porównywania z None używaj is zamiast ==

Operator równości == porównuje dwie wartości obiektów, podczas gdy operator tożsamości is porównuje dwie tożsamości obiektów. Pojęcia wartości i tożsamości opisałem w rozdziale 7. Dwa obiekty mogą przechowywać równoważne wartości, ale ponieważ są dwoma oddzielnymi obiektami, mają dwie oddzielne tożsamości. Za każdym razem, gdy porównujesz wartość z None, prawie zawsze możesz jednak używać operatora is zamiast ==.

W niektórych przypadkach wyrażenie spam == None może przyjmować wartość True nawet wtedy, gdy spam zawiera None. Może się to zdarzyć z powodu przeciążenia operatora == (zagadnienie to szczegółowo opisałem w rozdziale 17.). Jednak instrukcja spam is None sprawdzi, czy wartość w zmiennej spam rzeczywiście ma wartość None. Ponieważ None jest jedyną wartością typu danych NoneType, w dowolnym programie Pythona znajduje się tylko jeden obiekt None. Jeśli zmienna jest ustawiona na None, porównanie is None zawsze zwróci True. Szczegóły związane z przeciążaniem operatora == opisałem w rozdziale 17. Oto przykład tego zachowania:

```
>>> class someClass:
...     def __eq__(self, other):
...         if other is None:
...             return True
...
>>> spam = SomeClass()
>>> spam == None
True
>>> spam is None
False
```

Sytuacja, w której klasa przeciąża operator == w taki sposób, jaki pokazałem powyżej, jest rzadkością, ale idiomem Pythona stało się konsekwentne wykorzystywanie, tak na wszelki wypadek, konstrukcji is None zamiast == None.

Na koniec nie należy używać operatora is z wartościami True i False. Do porównywania wartości z True lub False używaj operatora ==, na przykład spam == True lub spam == False. Jeszcze bardziej powszechne jest całkowite opuszczanie operatora i wartości logicznej i pisanie kodu postaci if spam: lub if not spam: zamiast if spam==True: lub if spam == False:.

Formatowanie ciągów znaków

Ciągi znaków występują w prawie wszystkich programach komputerowych, bez względu na użyty język. Ten typ danych jest stosowany powszechnie, więc nie jest zaskoczeniem to, że istnieje wiele podejść do wykonywania działań na ciągach i formatowania ciągów. W tym podrozdziale przedstawiłem kilka sprawdzonych rozwiązań.

Kup ksi k Pole ksi k

Jeśli ciąg zawiera wiele lewych ukośników, używaj surowych ciągów znaków

Znaki ucieczki umożliwiają wstawianie do literałów znakowych tekstu, którego bez znaków ucieczki nie można by było wstawić. Na przykład w literale `'Zophie\'s chair'` potrzebne jest użycie lewego ukośnika (\), aby Python zinterpretował drugi apostrof jako część ciągu, a nie symbol oznaczający jego koniec. Ponieważ lewy ukośnik ma to specjalne znaczenie znaku ucieczki, to jeśli chcesz umieścić w ciągu rzeczywisty znak lewego ukośnika, powinieneś wprowadzić go jako \\.

Surowe ciągi znaków (ang. *raw strings*) to literały tekstowe oznaczone prefiksem `r`. Wewnątrz tych ciągów znaki lewego ukośnika nie są interpretowane jako symbole ucieczki. Zamiast tego są po prostu wstawiane do ciągu. Na przykład poniższy ciąg ścieżki do pliku w systemie Windows wymaga kilku znaków lewego ukośnika, co nie jest zbyt pythoniczne:

```
>>> # Przykład niepythoniczny
>>> print('Plik jest zapisany w katalogu C:\\Users\\Al\\Desktop\\Info\\Archive\\Spam')
Plik jest zapisany w katalogu C:\Users\Al\Desktop\Info\Archive\Spam
```

Zastosowany poniżej surowy ciąg (zwróć uwagę na prefiks `r`) tworzy tę samą wartość tekstową, ale jest znacznie bardziej czytelny:

```
>>> # Przykład pythoniczny
>>> print(r'Plik jest zapisany w katalogu C:\Users\Al\Desktop\Info\Archive\Spam')
Plik jest zapisany w katalogu C:\Users\Al\Desktop\Info\Archive\Spam
```

Surowe ciągi znaków nie są odrębnym typem tekstowych danych, są jedynie wygodnym sposobem wpisywania literałów tekstowych zawierających kilka znaków lewego ukośnika. Surowe ciągi znaków są często używane do wpisywania wyrażeń regularnych lub ścieżek do plików w systemie Windows. Wymienione teksty często zawierają kilka znaków lewego ukośnika. Stosowanie ucieczki dla każdego z nich pojedynczo, za pomocą symbolu \\, byłoby bardzo uciążliwe.

Formatowanie ciągów za pomocą f-stringów

Formatowanie ciągów lub *interpolacja ciągów* to proces tworzenia ciągów znaków zawierających inne ciągi znaków. Mają one w Pythonie długą historię. Najstarszy sposób łączenia ciągów polega na wykorzystaniu operatora `+`. W wyniku korzystania z tego sposobu tworzy się kod zawierający wiele apostrofów i plusów: `'Witaj, ' + name + '. Dzisiaj jest ' + day + ' i jest ' + weather + '.'`. Specyfikator konwersji `%s` uprościł nieco składnię: `'Witaj, %s. Dzisiaj jest %s i jest %s.' % (name, day, weather)`. Obie techniki pozwalają wstawić tekstowe wartości zmiennych `name`, `day` i `weather` do literałów tekstowych, co pozwala określić nową wartość ciągu, na przykład: `'Witaj, Al. Dzisiaj jest niedziela i jest słonecznie.'`.

Kup ksi k

Pole ksi k

Wprowadzenie metody `format()` wiązało się z dodaniem minijęzyka specyfikacji formatu (*https://docs.python.org/3/library/string.html#formatspec*), który bazuje na użyciu par nawiasów klamrowych `{}` w sposób podobny do specyfikatora konwersji `%s`. Jednak metoda ta jest nieco zawiła i może prowadzić do generowania nieczytelnego kodu, nie zalecam więc korzystania z niej.

Począwszy od Pythona 3.6 wprowadzono tzw. *f-stringi*, które oferują wygodniejszy sposób tworzenia ciągów zawierających inne ciągi. Podobnie jak surowe ciągi znaków są poprzedzone znakiem `r` przed pierwszym apostrofem, tak f-stringi są poprzedzone znakiem `f`. W f-stringach, aby wstawić ciągi przechowywane w zmiennych, możesz umieszczać nazwy tych zmiennych między nawiasami klamrowymi:

```
>>> name, day, weather = 'Al', 'niedziela', 'słonecznie'
>>> f'Witaj, {name}. Dzisiaj jest {day} i jest {weather}.'
'Witaj, Al. Dzisiaj jest niedziela i jest słonecznie.'
```

W nawiasach klamrowych można również umieszczać całe wyrażenia:

```
>>> width, length = 10, 12
>>> f'Pokój o wymiarach {width} na {length} ma powierzchnię {width * length}.'
'Pokój o wymiarach 10 na 12 ma powierzchnię 120.'
```

Jeśli chcesz użyć nawiasu wewnątrz f-stringu, możesz zastosować znak ucieczki w postaci dodatkowego nawiasu klamrowego:

```
>>> spam = 42
>>> f'Ta instrukcja wyświetla wartość w zmiennej spam: {spam}'
'Ta instrukcja wyświetla wartość w zmiennej spam: 42'
>>> f'Ta instrukcja wyświetla dosłownie nawiasy klamrowe: {{spam}}'
'Ta instrukcja wyświetla dosłownie nawiasy klamrowe: {spam}'
```

Dzięki możliwości umieszczenia wewnątrz ciągu wbudowanych nazw zmiennych i wyrażeń kod staje się bardziej czytelny niż przy użyciu starych sposobów formatowania ciągów.

Wymienione różne sposoby formatowania ciągów są sprzeczne z aforyzmem zen Pythona, że powinien istnieć jeden — a najlepiej tylko jeden — oczywisty sposób zrobienia czegoś. F-stringi są jednak ulepszeniem języka (moim zdaniem), a jak stanowi inna wytyczna, względy praktyczności są ważniejsze od względów czystości. Jeśli piszesz kod, który będzie uruchamiany w środowisku Pythona 3.6 lub nowszego, używaj f-stringów. Jeśli piszesz kod, który może być uruchamiany we wcześniejszych wersjach języka Python, stosuj konsekwentnie metodę `format()` lub specyfikatory konwersji `%s`.

Kup ksi k Pole ksi k

Tworzenie płytkich kopii list

Składnia *wycinków* (ang. *slice*) pozwala łatwo tworzyć nowe ciągi lub listy z istniejących. Aby zobaczyć, jak to działa, wprowadź w interaktywnej powłoce następujące instrukcje:

```
>>> 'Witaj, świecie!'[7:14] # Utworzenie ciągu na podstawie większego ciągu 'świecie'.
>>> 'Witaj, świecie!'[:5] # Utworzenie ciągu na podstawie większego ciągu.
'Witaj'
>>>['kot', 'pies', 'szczur', 'węgorz'][2:] # Utworzenie listy na podstawie większej listy.
['szczur', 'węgorz']
```

Dwukropek (:) oddziela początkowe i końcowe indeksy elementów, które mają się znaleźć na nowej liście. Jeśli pominiesz indeks początkowy przed dwukropkiem, jak w `'Hello, world!'. [:5]`, domyślnie indeks początkowy wyniesie 0. Jeśli pominiesz indeks końcowy za dwukropkiem, jak w `['kot', 'pies', 'szczur', 'węgorz'][2:]`, indeks końcowy zostanie domyślnie ustawiony na końcu listy.

Jeśli pominiesz obydwa indeksy, indeks początkowy będzie wynosił 0 (początek listy), a indeks końcowy zostanie ustawiony na końcu listy. W efekcie zostanie utworzona kopia listy:

```
>>> spam = ['kot', 'pies', 'szczur', 'węgorz']
>>> eggs = spam[:]
>>> eggs
['kot', 'pies', 'szczur', 'węgorz']
>>> id(spam) == id(eggs)
False
```

Zauważ, że tożsamości list `spam` i `eggs` są różne. Wiersz `eggs = spam[:]` tworzy płytką kopię listy `spam`, podczas gdy wiersz `eggs = spam` kopiuje tylko referencję do listy. Operator `[:]` wygląda jednak trochę dziwnie. Bardziej czytelny jest kod, który do utworzenia płytkiej kopii listy wykorzystuje funkcję `copy()` z modułu `copy`:

```
>>> # Przykład pythoniczny
>>> import copy
>>> spam = ['kot', 'pies', 'szczur', 'węgorz']
>>> eggs = copy.copy(spam)
>>> id(spam) == id(eggs)
False
```

Powinieneś rozpoznać tę dziwną składnię w przypadku, gdybyś natknął się na kod w Pythonie, który go używa. Nie polecam jednak stosowania go we własnym kodzie. Zapamiętaj, że zarówno operator `[:]`, jak i wywołanie `copy.copy()` tworzą płytkie kopie.

Pythoniczne sposoby korzystania ze słowników

Słowniki są podstawową strukturą danych w wielu programach w Pythonie. Decyduje o tym elastyczność, jaką zapewniają pary klucz-wartość (omówione dalej w rozdziale 7.), pozwalające zmapować jeden fragment danych na drugi. Z tego powodu warto zapoznać się z kilkoma idiomami dotyczącymi słowników, często stosowanymi w kodzie Pythona.

Aby uzyskać więcej informacji na temat słowników, zapoznaj się z niezwykłym referatem programisty Pythona Brandona Rhodesa na temat słowników i ich działania wygłoszonym na konferencji PyCon 2010 i zatytułowanym *The Mighty Dictionary*. Jest on dostępny pod adresem *https://invpy.com/*. Warto również sięgnąć do referatu *The Dictionary Even Mightier* z konferencji PyCon 2017, który jest dostępny pod adresem *https://invpy.com/dictionaryevenmightier*.

Używaj ze słownikami wywołań get() i setdefault()

Próba uzyskania dostępu w słowniku do klucza, który nie istnieje, spowoduje błąd `KeyError`, więc programiści, aby uniknąć tej sytuacji, często piszą niepythoniczny kod w następujący sposób:

```
>>> # Przykład niepythoniczny
>>> numberOfPets = {'psy': 2}
>>> if 'koty' in numberOfPets: # Sprawdzenie, czy istnieje klucz 'koty'.
...     print('Mam ', numberOfPets['koty'], ' kotów.')
... else:
...     print('Mam 0 kotów.')
...
Mam 0 kotów.
```

Ten kod sprawdza, czy w słowniku `numberOfPets` istnieje jako klucz ciąg `'koty'`. Jeśli tak, w wywołaniu funkcji `print()`, w celu utworzenia komunikatu dla użytkownika, następuje odwołanie do `numberOfPets['koty']`. Jeśli nie, inne wywołanie funkcji `print()` drukuje komunikat bez sięgania do `numberOfPets['koty']`. Dzięki temu ten kod nie zgłasza wyjątku `KeyError`.

Ten wzorzec jest tak powszechny, że do słowników wprowadzono metodę `get()`, pozwalającą określić wartość domyślną, która zostanie zwrócona, gdy klucz w słowniku nie istnieje. Poniższy, pythoniczny kod jest odpowiednikiem poprzedniego przykładu:

```
>>> # Przykład pythoniczny
>>> numberOfPets = {'psy': 2}
>>> print('Mam ', numberOfPets.get('koty', 0), 'kotów.')
Mam 0 kotów.
```


Kup ksi k Pole ksi k

Wywołanie `numberOfPets.get('koty', 0)` sprawdza, czy w słowniku `numberOfPets` istnieje klucz `"koty"`. Jeśli tak, wywołanie metody zwraca wartość dla klucza `"koty"`. Jeśli nie, zamiast tego zwraca drugi argument.

Korzystanie z metody `get()` w celu ustalenia wartości domyślnej dla nieistniejących kluczy jest krótsze i bardziej czytelne niż użycie instrukcji `if-else`.

Poza tym możesz chcieć ustawić wartość domyślną, jeśli klucz nie istnieje. Na przykład jeśli słownik `numberOfPets` nie ma klucza `"koty"`, instrukcja `numberOfPets['koty'] += 10` spowoduje błąd `KeyError`. Mógłbyś dodać kod, który sprawdza istnienie klucza, i w przypadku jego braku ustawia wartość domyślną:

```
>>> # Przykład niepythoniczny
>>> numberOfPets = {'psy': 2}
>>> if 'koty' not in numberOfPets:
...     numberOfPets['koty'] = 0
...
>>> numberOfPets['koty'] += 10
>>> numberOfPets['koty']
10
```

Ponieważ ten wzorzec również jest powszechny, słowniki mają bardziej pythoniczną metodę `setdefault`. Poniższy kod jest równoważny z poprzednim przykładem:

```
>>> # Przykład pythoniczny
>>> numberOfPets = {'psy': 2}
>>> numberOfPets.setdefault('koty', 0) # Jeśli klucz 'koty' istnieje, nie rób niczego.
0
>>> numberOfPets['koty'] += 10
>>> numberOfPets['koty']
10
```

Jeśli piszesz instrukcje `if`, które sprawdzają, czy istnieje klucz w słowniku, i ustawiasz wartość domyślną, jeśli klucz nie istnieje, użyj zamiast nich metody `setdefault()`.

Użyj dla wartości domyślnych klasy collections.defaultdict

Aby całkowicie wyeliminować błędy `KeyError`, możesz zastosować klasę `collections.defaultdict`. Umożliwia ona tworzenie domyślnego słownika przez zaimportowanie modułu `collections` i wywoływanie funkcji `collections.defaultdict()` oraz przekazanie typu danych do wykorzystania w roli wartości domyślnej. Na przykład jeśli do wywołania `collections.defaultdict()` przekażesz `int`, możesz stworzyć podobny do słownika obiekt, który wykorzystuje `0` dla wartości domyślnej w przypadku nieistniejących kluczy. Wprowadź w interaktywnej powłoce następujące instrukcje:

Kup książkę Poleć książkę

```
>>> import collections
>>> scores = collections.defaultdict(int)
>>> scores
defaultdict(<class 'int'>, {})
>>> scores['Al'] += 1 # Nie ma potrzeby wcześniejszego ustawiania wartości dla klucza 'Al'.
>>> scores
defaultdict(<class 'int'>, {'Al': 1})
>>> scores['Zophie'] # Nie ma potrzeby wcześniejszego ustawiania wartości dla klucza 'Zophie'.
0
>>> scores['Zophie'] += 40
>>> scores
defaultdict(<class 'int'>, {'Al': 1, 'Zophie': 40})
```

Zwróć uwagę, że nie wywołujesz funkcji `int()`, tylko ją przekazujesz, więc w wywołaniu `collections.defaultdict(int)` po `int` nie umieszczasz nawiasów. Możesz również przekazać `list`, aby jako wartości domyślnej użyć pustej listy. Wprowadź w interaktywnej powłoce następujące instrukcje:

```
>>> import collections
>>> booksReadBy = collections.defaultdict(list)
>>> booksReadBy['Al'].append('Oryx and Crake')
>>> booksReadBy['Al'].append('American Gods')
>>> len(booksReadBy['Al'])
2
>>> len(booksReadBy['Zophie']) # Wartość domyślna jest pustą listą.
0
```

Jeśli potrzebujesz wartości domyślnej dla każdego możliwego klucza, to znacznie łatwiej jest używać wywołania `collections.defaultdict()`, niż korzystać ze zwykłego słownika i stale wywoływać metodę `setdefault()`.

Używaj słowników zamiast instrukcji switch

W takich językach jak Java występuje instrukcja `switch`, która jest rodzajem instrukcji `if-elif-else` uruchamiającej kod na podstawie jednej z wielu wartości zmiennej. Python nie zawiera instrukcji `switch`, więc jego programiści czasami piszą kod taki jak w poniższym przykładzie, który uruchamia inną instrukcję przypisania, na podstawie jednej z wielu wartości zmiennej `season`:

```
# We wszystkich poniższych warunkach if i elif występuje porównanie "season ==":
if season == 'Zima':
    holiday = 'Nowy Rok'
elif season == 'Wiosna':
    holiday = 'Święto Pracy'
elif season == 'Lato':
    holiday = 'Boże Ciało'
elif season == 'Jesień':
    holiday = 'Wszystkich Świętych'
else:
    holiday = 'Dzień wolny'
```


Kup ksi k Pole ksi k

Powyższy kod niekoniecznie jest niepythoniczny, ale jest nieco rozwlekły. Domyślnie instrukcje `switch` w Javie mają mechanizm „fall-through”, który wymaga, aby każdy blok kończył się instrukcją `break`. W przeciwnym razie sterowanie przechodzi do następnego bloku. Zapominanie o konieczności dodania tej instrukcji `break` jest częstym źródłem błędów. Jednak wszystkie instrukcje `if-elif` w pokazanym przykładzie kodu w Pythonie mogą być powtarzalne.

Niektórzy programiści Pythona, zamiast używać instrukcji `if-elif`, wolą skonfigurować słownik. Poniższy zwięzły i pythoniczny kod jest odpowiednikiem poprzedniego przykładu:

```
holiday = {'Zima'  : 'Nowy Rok',
           'Wiosna': 'Święto Pracy',
           'Lato'  : 'Boże Ciało',
           'Jesień': 'Wszystkich Świętych'}.get(season, 'Dzień wolny')
```

Cały ten kod to jedna instrukcja przypisania. Wartość przechowywana w zmiennej `holiday` jest zwracana z wywołania metody `get()`. Jest to wartość spod klucza ustawionego w zmiennej `season`. Jeśli klucz `season` nie istnieje, `get()` zwraca `"Dzień wolny"`. Dzięki wykorzystaniu słownika kod staje się bardziej zwięzły, ale jednocześnie trudniejszy do czytania. To, czy użyjesz tej konwencji, zależy tylko od Ciebie.

Wyrażenia warunkowe: „brzydki” operator trójargumentowy Pythona

Operatory trójargumentowe (oficjalnie nazywane *wyrażeniami warunkowymi* lub czasami *trójargumentowymi wyrażeniami wyboru*) przypisują wartość wyrażenia do jednej z dwóch wartości na podstawie warunku. Zwyczajnie można to zrobić za pomocą pythonicznej instrukcji `if-else`:

```
>>> # Przykład pythoniczny
>>> condition = True
>>> if condition:
...     message = 'Dostęp przyznany'
... else:
...     message = 'Blokada dostępu'
...
>>> message
'Dostęp przyznany'
```

Przymiotnik *trójargumentowy* oznacza po prostu trzy wejścia, ale w programowaniu jest synonimem *wyrażenia warunkowego*. Wyrażenia warunkowe pozwalają również na tworzenie bardziej zwięzłych, jednolinijkowych instrukcji

Kup ksi k Pole ksi k

dla kodu, który pasuje do tego wzorca. W Pythonie są one implementowane za pomocą dziwnego układu słów kluczowych `if` i `else`:

```
>>> valueIfTrue = 'Dostęp przyznany'
>>> valueIfFalse = 'Blokada dostępu'
>>> condition = True
>>> message = valueIfTrue if condition else valueIfFalse ❶
>>> message
'Dostęp przyznany'
>>> print(valueIfTrue if condition else valueIfFalse) ❷
'Dostęp przyznany'
>>> condition = False
>>> message = valueIfTrue if condition else valueIfFalse
>>> message
'Blokada dostępu'
```

Wyrażenie `valueIfTrue if condition else valueIfFalse` ❶ przyjmuje wartość `valueIfTrue`, jeśli zmienna `condition` ma wartość `True`. Gdy zmienna `condition` ma wartość `False`, wyrażenie przyjmuje wartość `valueIfFalse`. Guido van Rossum żartobliwie opisał swój projekt składni jako „celowo brzydki". W większości języków z operatorem trójargumentowym w wyrażeniu najpierw występuje warunek, następnie wartość odpowiadająca warunkowi `True`, a za nią wartość dla warunku `False`. Wyrażenia warunkowe mogą być używane wszędzie tam, gdzie można użyć wyrażenia lub wartości, w tym jako argument wywołania funkcji ❷.

Dlaczego w Pythonie 2.5 wprowadzono tę składnię, mimo że łamie ona pierwszą wytyczną, że „piękne jest lepsze niż brzydkie"? Niestety, pomimo tego, że operatory trójargumentowe są trochę nieczytelne, wielu programistów ich używa i chciało, aby Python obsługiwał tę składnię. Aby stworzyć rodzaj operatora trójargumentowego, możliwe jest także nadużycie „zwarcia operatorów logicznych". Wyrażenie `condition and valueIfTrue or valueIfFalse` przyjmuje wartość `valueIfTrue`, jeśli `condition` ma wartość `True`, lub `valueIfFalse`, jeśli `condition` ma wartość `False` (z wyjątkiem jednego ważnego przypadku). Wprowadź w interaktywnej powłoce następujące instrukcje:

```
>>> # Przykład niepythoniczny
>>> valueIfTrue = 'Dostęp przyznany'
>>> valueIfFalse = 'Blokada dostępu'
>>> condition = True
>>> condition and valueIfTrue or valueIfFalse
'Dostęp przyznany'
```

Pseudooperator trójargumentowy postaci `condition and valueIfTrue or valueIfFalse` ma subtelny błąd: jeśli `valueIfTrue` jest jedną z wartości: `0`, `False`, `None`, `""`, to pomimo że zmienna `condition` ma wartość `True`, wyrażenie nieoczekiwanie przyjmuje wartość `valueIfFalse`.

Kup książkę Poleć książkę

Programiści nadal jednak używali tego fałszywego operatora trójargumentowego, a pytanie „Dlaczego Python nie ma operatora trójargumentowego?" stało się jednym z najczęstszych pytań zadawanych głównym programistom języka Python. Wyrażenia warunkowe zostały utworzone po to, by programiści przestali prosić o wprowadzenie operatora trójargumentowego i aby przestali używać podatnego na błędy pseudooperatora. Wyrażenia warunkowe są jednak na tyle brzydkie, że zniechęcają programistów do korzystania z nich. Chociaż „piękne jest lepsze niż brzydkie", „brzydki" operator trójargumentowy w Pythonie jest przykładem przypadku, gdy względy praktyczne biorą górę nad względami czystości.

Wyrażenia warunkowe nie są do końca pythoniczne, ale nie są też niepythoniczne. Jeśli ich używasz, staraj się unikać zagnieżdżania wyrażeń warunkowych wewnątrz innych wyrażeń warunkowych:

```
>>> # Przykład niepythoniczny
>>> age = 30
>>> ageRange = 'dziecko' if age < 13 else 'nastolatek' if age >= 13 and age < 18
else 'dorosły'
>>> ageRange
'dorosły'
```

Zagnieżdżone wyrażenia warunkowe są dobrym przykładem na to, że długi pojedynczy wiersz kodu, choć jest technicznie poprawny, może być frustrujący i nieczytelny.

Korzystanie z wartości zmiennych

Często powstaje potrzeba sprawdzania i modyfikowania wartości przechowywanych w zmiennych. W Pythonie jest na to kilka sposobów. Spójrzmy na kilka przykładów.

Operatory przypisania i operatory porównania

Gdy musisz sprawdzić, czy liczba mieści się w określonym zakresie, możesz użyć logicznego operatora `and` w ten oto sposób:

```
# Przykład niepythoniczny
if 42 < spam and spam < 99:
```

Python pozwala jednak na tworzenie łańcuchów operatorów porównania. Dzięki temu nie trzeba używać operatora `and`. Poniższy kod jest odpowiednikiem poprzedniego przykładu:

```
# Przykład pythoniczny
if 42 < spam < 99:
```


Kup ksi k Pole ksi k

To samo dotyczy tworzenia łańcucha operatora przypisania =. W jednym wierszu kodu możesz ustawić wiele zmiennych na tę samą wartość:

```
>>> # Przykład pythoniczny
>>> spam = eggs = bacon = 'string'
>>> print(spam, eggs, bacon)
string string string
```

Aby sprawdzić, czy wszystkie trzy z tych zmiennych są takie same, można użyć operatora `and` lub, w celu sprawdzenia równości, po prostu utworzyć łańcuch operatora porównania ==.

```
>>> # Przykład pythoniczny
>>> spam = eggs = bacon = 'string'
>>> spam == eggs == bacon == 'string'
True
```

Łańcuchy operatorów to prosty, ale przydatny skrót w Pythonie. Jednak użycie ich niepoprawnie może powodować problemy. Kilka przypadków, gdy nieprawidłowe posługiwanie się łańcuchami operatorów może powodować nieoczekiwane błędy w kodzie, pokazałem w rozdziale 8.

Sprawdzanie, czy zmienna jest jedną z wielu wartości

Czasami może wystąpić odwrotność sytuacji opisanej w poprzednim punkcie: chcesz sprawdzić, czy określona zmienna jest jedną z wielu możliwych wartości. Możesz to zrobić za pomocą operatora `or`, jak w wyrażeniu `spam == 'kot' or spam == 'pies' or spam == 'łoś'`. Wszystkie te powielone fragmenty "`spam ==`" sprawiają, że to wyrażenie jest nieco nieporęczne.

Zamiast tego możesz umieścić wiele wartości w krotce i sprawdzić za pomocą operatora `in`, czy wartość zmiennej istnieje w tej krotce, jak w poniższym przykładzie:

```
>>> # Przykład pythoniczny
>>> spam = 'kot'
>>> spam in ('kot', 'pies', 'łoś')
True
```

Ten idiom nie tylko jest łatwiejszy do zrozumienia, ale także, sądząc po wynikach uzyskanych za pomocą `timeit`, nieco szybszy.

Kup ksi k Pole ksi k

Podsumowanie

Wszystkie języki programowania mają własne idiomy i najlepsze rozwiązania. Ten rozdział koncentruje się na konkretnych sposobach, na jakie programiści Pythona piszą kod „pythonicznie" — czyli tak, aby jak najlepiej wykorzystać składnię Pythona.

Sednem kodu pythonicznego jest 20 aforyzmów zen Pythona. Są one ogólnymi wytycznymi dotyczącymi pisania kodu w Pythonie. Te aforyzmy to opinie, które nie są absolutnie niezbędne do pisania kodu w Pythonie, ale warto o nich pamiętać.

Znaczące wcięcia w Pythonie (nie mylić ze znaczącymi odstępami) wywołują najwięcej protestów ze strony nowych programistów Pythona. Chociaż wcięcia są wykorzystywane w prawie wszystkich językach programowania w celu poprawy czytelności kodu, w Pythonie są one obowiązkowe i zastępują bardziej popularne nawiasy klamrowe stosowane w innych językach.

Mimo że wielu programistów Pythona używa dla pętli `for` konwencji `range(len())`, czystsze podejście do uzyskiwania indeksu i wartości podczas iteracji przez sekwencję oferuje funkcja `enumerate()`. Podobnie instrukcja `with`, w porównaniu z ręcznym wywoływaniem `open()` i `close()`, jest czystszym sposobem obsługi plików, stwarzającym mniej okazji do popełnienia błędów. Zastosowanie instrukcji `with` daje pewność wywołania metody `close()` za każdym razem, gdy sterowanie wyjdzie poza blok instrukcji `with`.

W Pythonie dostępnych jest kilka sposobów interpolacji ciągów znaków. Pierwszy z nich polega na użyciu specyfikatora konwersji `%s` do oznaczenia miejsc, w których w oryginalnym ciągu powinny być zawarte wartości zmiennych. Nowoczesny sposób, dostępny począwszy od Pythona 3.6, polega na użyciu f-stringów. W f-stringach ciągi są poprzedzone literą `f` i korzystają z nawiasów klamrowych do zaznaczenia miejsc występowania wartości zmiennych (lub całych wyrażeń).

Składnia `[:]` do tworzenia płytkich kopii list wygląda nieco dziwnie i niekoniecznie jest pythoniczna, ale stała się powszechnym sposobem szybkiego tworzenia płytkiej kopii listy.

Słowniki mają metody `get()` i `setdefault()`, służące do obsługi nieistniejących kluczy. Alternatywnie można skorzystać z wywołania `collections.defaultdict`, pozwalającego użyć wartości domyślnych dla nieistniejących kluczy. Ponadto, chociaż w Pythonie nie ma instrukcji `switch`, użycie słowników stanowi zwięzły sposób implementowania jej odpowiednika bez użycia kilku instrukcji `if-elif-else`. Do oceny wyrażeń przyjmujących dwie wartości w zależności od warunku można użyć operatorów trójargumentowych.

Łańcuch operatorów `==` pozwala sprawdzić równość wielu zmiennych, podczas gdy operator `in` umożliwia sprawdzenie, czy wartość zmiennej jest jedną z wielu możliwych.

W tym rozdziale opisałem kilka idiomów języka Python. Zamieściłem też wskazówki dotyczące pisania kodu bardziej pythonicznego. W następnym rozdziale dowiesz się o niektórych pythonowych haczykach i pułapkach, w które wpadają początkujący.

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

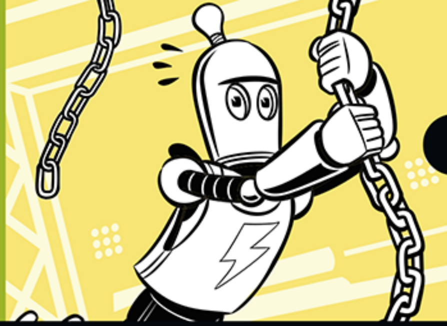
GRUPY HELION

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl



PYTHON. OD POCZĄTKU TWÓRZ CZYSTY, DOSKONALE PYTHONICZNY KOD!

Programistów Pythona ograniczają tylko wyobraźnia i technologia. Nic dziwnego, że rzesza osób zafascynowanych Pythonem stale rośnie, podobnie jak liczba adeptów programowania w tym języku. Po ugruntowaniu podstaw tworzenia kodu można już pisać całkiem funkcjonalne aplikacje, jednak prawdziwa moc Pythona i satysfakcja z pracy z nim objawiają się dopiero po opanowaniu nieco bardziej złożonych zagadnień. Jeśli więc znasz podstawową składnię i najważniejsze zasady rządzące Pythonem, czas na naukę pisania czystego, czytelnego i łatwego do utrzymania kodu — kodu pythonicznego!

Dzięki tej książce przyswoisz najlepsze zasady konfigurowania środowiska programistycznego i praktyki programistyczne poprawiające czytelność kodu. Znajdziesz tu mnóstwo przydatnych wskazówek dotyczących posługiwania się wierszem polecenia i takimi narzędziami jak formatery kodu, kontrolery typów, lintery, a nawet systemy kontroli wersji. Od strony praktycznej poznasz techniki organizacji kodu i tworzenia jego dokumentacji. Nie brak też zaawansowanych zagadnień, jak pomiary wydajności kodu czy analiza algorytmów Big O. Sporo miejsca poświęcono również pythonicznemu paradygmatowi programowania zorientowanego obiektowo. Dowiesz się więc, jak prawidłowo pisać klasy, korzystać z mechanizmów dziedziczenia i czym są metody dunder.

To znakomity przewodnik na drodze, którą musi pokonać początkujący, aby stać się profesjonalnym programistą Pythona.

W książce między innymi:

- czym jest dobry styl programowania
- automatyczne formatowanie kodu w Pythonie
- typowe źródła błędów i ich wykrywanie
- organizacja plików kodu w projektach
- programowanie funkcyjne w Pythonie
- profilowanie wydajności kodu

Al Sweigart — znakomity programista, obdarzony niezwykłym talentem dydaktycznym. Autor popularnych kursów programowania dla dzieci i dorosłych, w tym uwielbianego przez początkujących Udemy Python. Napisał też kilka popularnych podręczników do nauki programowania w Pythonie i Scratchu.

Helion

helion.pl

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

Sprawdź nasze szkolenia!

HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-8338-8

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 79,00 zł