



Python

dla profesjonalistów

Debugowanie, testowanie i utrzymywanie kodu

—

Kristian Rother

Helion 

Apress

Tytuł oryginału: Pro Python Best Practices: Debugging, Testing and Maintenance

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-3802-9

Original edition copyright © 2017 by Kristian Rother
All rights reserved.

Polish edition copyright © 2018 by HELION SA
All rights reserved.

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

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

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

Autor oraz 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/pytpro>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

	O autorze	13
	O recenzencie technicznym	15
	Podziękowania	17
	Przedmowa	19
Rozdział 1.	Wprowadzenie	21
	Lekcja pokory	21
	Najlepsze praktyki w Pythonie	21
	Pochodzenie najlepszych praktyk	23
	Hacking	23
	Inżynieria programowania	25
	Agile	25
	Software Craftsmanship	25
	Dla kogo jest ta książka?	26
	O czym jest ta książka?	27
	Część I. „Debugowanie”	27
	Część II. „Automatyczne testowanie”	27
	Część III. „Utrzymanie”	27
	Dalsze korzyści	28
	Gra MazeRun	28
	Jak korzystać z tej książki?	29
	Instalacja Pythona 3	29
	Instalacja biblioteki Pygame	29
	Instalacja edytora tekstu	30
	Pobranie kodu źródłowego przykładów	30

Część I	Debugowanie	31
Rozdział 2.	Wyjątki w Pythonie	33
	Wyjątki są defektami, o których istnieniu wiemy	33
	Czytanie kodu	35
	Błędy typu SyntaxError	35
	Najlepsze praktyki debugowania wyjątków SyntaxError	37
	Analiza komunikatów o błędach	37
	Typ błędu	38
	Opis błędu	38
	Ślad	39
	Dedukcja	40
	Przechwytywanie wyjątków	40
	Najlepsze praktyki debugowania wyjątków IOError	41
	Błędy i defekty	42
	Skąd się biorą defekty?	42
	Poprawny kod	43
	Najlepsze praktyki	44
Rozdział 3.	Błędy semantyczne w Pythonie	45
	Porównywanie rzeczywistych danych wyjściowych z oczekiwanymi	46
	Defekty	47
	Defekty przypisania zmiennych	48
	Wielokrotne inicjowanie	48
	Przypadkowe przypisanie	49
	Przypadkowe porównania	49
	Nieprawidłowe zmienne w wyrażeniu	49
	Przestawione literały w wyrażeniu	50
	Defekty indeksowania	51
	Tworzenie nieprawidłowych indeksów	51
	Użycie nieprawidłowych indeksów	52
	Defekty w instrukcjach przepływu sterowania	53
	Defekty w wyrażeniach logicznych	54
	Defekty związane z wcięciami	54
	Defekty w używaniu funkcji	55
	Pomijanie wywołania funkcji	55
	Brak instrukcji return	55
	Brak przechowywania zwracanej wartości	55
	Propagacja błędów	56
	Najlepsze praktyki	57
Rozdział 4.	Debugowanie metodą naukową	59
	Stosowanie metody naukowej	60
	Odtwarzanie błędu	61
	Automatyzowanie błędu	62
	Izolowanie defektu	63
	Strategia rozbierania	63
	Strategia wyszukiwania binarnego	64

Uzyskiwanie pomocy	65
Czas na przerwę	65
Wyjaśnij problem komuś innemu	65
Programowanie w parach	65
Przeglądy kodu	65
Czytanie	66
Sprzątanie	68
Metoda naukowa i inne najlepsze praktyki	68
Najlepsze praktyki	68
Rozdział 5. Debugowanie za pomocą instrukcji print	71
Diagnozowanie, czy kod był uruchamiany	73
Wyświetlanie zawartości zmiennych	73
Estetyczne wyświetlanie struktur danych	74
Upraszczenie danych wejściowych	74
Zacznij od minimalnego wejścia	75
Stopniowe dodawanie większej ilości danych wejściowych	76
Włączanie i wyłączanie wyświetlania wyjścia	78
Kompletny kod	79
Plusy i minusy używania instrukcji print	79
Najlepsze praktyki	80
Rozdział 6. Debugowanie z funkcjami introspekcji	81
Kodowanie eksploracyjne w IPythonie	82
Eksploracja plików i katalogów	83
Przegląd poleceń powłoki IPython	83
Odkrywanie przestrzeni nazw	84
Eksploracja przestrzeni nazw za pomocą polecenia dir()	85
Eksploracja przestrzeni nazw obiektów	86
Eksploracja atrybutów w programie Pythona	87
Alternatywy instrukcji dir w IPythonie	87
Mechanika przestrzeni nazw	88
Python używa przestrzeni nazw dla własnych funkcji	88
Modyfikowanie przestrzeni nazw	88
Przestrzenie nazw i zasięg lokalny	89
Przestrzenie nazw są podstawową właściwością Pythona	90
Używanie samodokumentujących się obiektów	90
Dostęp do ciągów dokumentacyjnych za pomocą instrukcji help()	91
Opisy obiektów w IPythonie	91
Analizowanie typów obiektów	92
Sprawdzanie tożsamości obiektu	93
Sprawdzanie egzemplarzy i podklas	93
Praktyczne wykorzystanie introspekcji	93
Znajdowanie literówek za pomocą introspekcji	94
Łączenie funkcji introspekcji	95
Introspekcja w dużych i małych programach	95
Najlepsze praktyki	96

Rozdział 7. Korzystanie z interaktywnego debugera	97
Interaktywny debugger — ipdb	98
Instalowanie ipdb	99
Uruchamianie debugera	99
Uruchamianie ipdb z wiersza polecenia	99
Uruchamianie ipdb z poziomu programu	99
Debugowanie post mortem	100
Uruchamianie debugera w odpowiedzi na wyjątki	101
Naprawa defektu	102
Komendy w wierszu poleceń debugera	102
Inspekcja zmiennych	102
Ocena wartości wyrażeń Pythona	102
Krokowe uruchamianie kodu	102
Wznawianie działania programu	103
Używanie pułapek	103
Przeglądanie i usuwanie pułapek	104
Pułapki warunkowe	104
Konfigurowanie ipdb	105
Przykład sesji ipdb	105
Dodawanie funkcji sterowania grą	105
Krokowe uruchamianie kodu	106
Usuwanie defektu	108
To działa!	109
Czy teraz program nie ma defektów?	109
Inne narzędzia do debugowania	109
pdb — debugger Pythona	109
Środowisko IDE PyCharm	109
ipdbplugin	110
pudb	110
wdb	111
Pasek narzędzi Django Debug	111
cProfile	111
Najlepsze praktyki	111
Część II Automagiczne testowanie	113
Rozdział 8. Pisanie automatycznych testów	115
Instalacja frameworka py.test	116
Pisanie funkcji testowej	116
Uruchamianie testów	117
Pisanie testu, który nie przechodzi	117
Spraw, aby test przeszedł	118
Testy pomyślne a testy niepomyślne	118
Pisanie oddzielnych funkcji testowych	119
Asercje dostarczają przydatnych danych wyjściowych	120
Testowanie występowania wyjątków	121
Przypadki brzegowe	122

Złożone przypadki brzegowe	124
Korzyści wynikające z automatycznego testowania	125
Inne frameworki testowe w Pythonie	126
unittest	126
nose	126
doctest	126
Pisanie bloku <code>__main__</code>	126
Najlepsze praktyki	126
Rozdział 9. Organizowanie danych testowych	129
Używanie fikstur	130
Parametr <code>scope</code>	131
Parametryzacja testów	132
Wiele parametrów	133
Fikstury z parametrami	133
Makiety	134
Testowanie plików wynikowych	135
Sprzątanie po testach	136
Używanie plików tymczasowych	136
Porównywanie plików wynikowych z danymi testowymi	137
Najlepsze praktyki testowania dużych plików	138
Generowanie losowych danych testowych	138
Gdzie przechowywać dane testowe?	139
Moduły danych testowych	139
Katalogi danych testowych	139
Bazy danych testowych	139
Najlepsze praktyki	139
Rozdział 10. Pisanie zestawu testów	141
Moduły testowe	142
Klasy testów	143
Refaktoryzacja funkcji testowych	144
Fikstury w klasach testowych	146
W jaki sposób testy znajdują testowany kod?	146
Wiele pakietów testowych	147
Automatyczne wykrywanie testów	148
Uruchamianie zestawu testów	148
Częściowe uruchomienie	149
Obliczanie pokrycia testami	151
Zestaw testów wymaga utrzymania	153
Najlepsze praktyki	154
Rozdział 11. Najlepsze praktyki testowania	155
Rodzaje automatycznych testów	156
Testy jednostkowe	156
Testy integracyjne	157
Testy akceptacyjne	157
Testy regresji	157

Testy wydajności	158
Optymalizacja wydajności	159
Podejście „najpierw test”	160
Pisanie testów według specyfikacji	161
Pisanie testów według defektów	162
Rozwój oprogramowania sterowany testami (TDD)	162
Zalety automatycznego testowania	163
Testowanie oszczędza czas	163
Testowanie dodaje precyzji	163
Dzięki testowaniu współpraca staje się łatwiejsza	164
Ograniczenia automatycznego testowania	164
Testowanie wymaga sprawdzalnego kodu	164
Testowanie nie działa dobrze w przypadku projektów, które szybko się zmieniają	164
Testowanie nie udowadnia poprawności	164
Programy trudne do testowania	165
Liczby losowe	165
Graficzne interfejsy użytkownika	165
Dane wyjściowe złożone lub o dużej objętości	166
Współbieżność	166
Sytuacje, gdy automatyczne testy zawodzą	166
Inne możliwości dla automatycznego testowania	167
Tworzenie prototypów	167
Przeglądy kodu	167
Listy kontrolne	167
Procesy promujące poprawność	168
Wnioski	168
Najlepsze praktyki	169

Część III Utrzymanie171

Rozdział 12 . Kontrola wersji 173

Wprowadzenie do pracy z systemem git	174
Tworzenie repozytorium	174
Dodawanie plików do repozytorium	175
Śledzenie zmian w plikach	176
Przenoszenie i usuwanie plików	177
Odrzucanie zmian	177
Przeglądanie historii kodu	178
Pobieranie starszych commitów	179
Powrót do najświeższego commita	179
Publikowanie kodu w serwisie GitHub	179
Rozpoczynanie projektu w serwisie GitHub	180
Korzystanie z serwisu GitHub z pozycji pojedynczego programisty	181
Praca w projektach rozpoczętych przez innych	181
Projekty z wieloma programistami	181
Scalanie zmian wprowadzonych przez dwie osoby	182
Żądania pobrania	182

Korzystanie z gałęzi	182
Scalanie gałęzi	183
Konfigurowanie systemu git	184
Ignorowanie plików	184
Ustawienia globalne	185
Przykłady użycia	185
Dwadzieścia znaków: mały projekt o małym ruchu	186
Python: wielki projekt z codziennymi commitami	186
grep: projekt długoterminowy	187
Inne systemy kontroli wersji	188
Mercurial	188
Subversion (SVN)	188
CVS (Concurrent Versions Software)	188
Bitbucket	188
Sourceforge	188
Najlepsze praktyki	189
Rozdział 13. Konfigurowanie projektu w Pythonie	191
Tworzenie struktury projektu za pomocą narzędzia pyscaffold	192
Instalacja narzędzia pyscaffold	192
Typowe katalogi w projekcie Pythona	193
Katalogi utworzone przez pyscaffold	193
Katalogi nieutworzone przez pyscaffold	194
Pliki	195
Pliki utworzone przez pyscaffold	195
Pliki, które nie są tworzone przez pyscaffold	196
Ustawianie numerów wersji programu	196
Zarządzanie środowiskiem projektu Pythona za pomocą virtualenv	197
Instalacja narzędzia virtualenv	198
Podłączanie projektu do środowiska virtualenv	198
Praca z projektem virtualenv	199
Instalowanie pakietów w środowisku virtualenv	199
Opuszczanie sesji środowiska virtualenv	200
Konfigurowanie uruchamiania i dezaktywacji środowiska virtualenv	200
Instalowanie Pygame z virtualenv	202
Najlepsze praktyki	202
Rozdział 14. Porządkowanie kodu	203
Kod zorganizowany i niezorganizowany	203
Entropia oprogramowania: przyczyny niezorganizowanego kodu	204
Jak rozpoznać niezorganizowany kod?	206
Czytelność	206
Niedoskonałości strukturalne	206
Redundancja	207
Słabości projektu	208
Porządkowanie instrukcji Pythona	208
Pogrupuj instrukcje importu	209
Pogrupuj stałe	209

Usun niepotrzebne wiersze	210
Zastosuj opisowe nazwy zmiennych	210
Idiomatyczny kod Pythona	211
Refaktoryzacja	211
Wyodrębnianie funkcji	212
Tworzenie prostego interfejsu wiersza polecenia	213
Podział programów na moduły	213
Uporządkowany kod	214
PEP8 i pylint	215
Komunikaty ostrzegawcze	215
Ocena punktowa kodu	215
Zrób tak, żeby działało, zrób to dobrze, zrób tak, żeby działało szybko	216
Zrób tak, żeby działało	216
Zrób to dobrze	217
Zrób tak, żeby działało szybko	217
Przykłady dobrze zorganizowanego kodu	217
Najlepsze praktyki	218
Rozdział 15. Dekompozycja zadań programistycznych	219
Dekompozycja zadań programowania jest trudna	219
Proces dekompozycji zadań programowania	220
Napisz historyjkę użytkownika	221
Dodaj szczegóły do opisu	222
Kryteria akceptacji	222
Opisy przypadków użycia	223
Sprawdź wymagania niefunkcjonalne	224
Identyfikowanie problemów	225
Niepełne informacje	225
Wiedza specjalistyczna	225
Zmiany istniejącego kodu	226
Przewidywanie przyszłych zmian	226
Wybór architektury	227
Identyfikowanie komponentów programu	229
Implementacja	231
Inne narzędzia planowania	233
Plan projektu na jednej stronie	233
Śledzenie spraw	233
Kanban	233
Najlepsze praktyki	234
Rozdział 16. Statyczne typowanie w języku Python	235
Słabe strony dynamicznego typowania	236
Sygnatury funkcji	236
Granice wartości	236
Semantyczne znaczenie typów	236
Typy złożone	237
Czy w Pythonie jest możliwe silniejsze typowanie?	237
Asercje	237

NumPy	239
Bazy danych	240
Integracja kodu w języku C	241
Cython	242
Wskazówki typowania	243
mypy	244
Której metody kontroli typów używać?	245
Najlepsze praktyki	247
Rozdział 17. Dokumentacja	249
Dla kogo piszemy dokumentację?	249
Sphinx: narzędzie do tworzenia dokumentacji dla języka Python	250
Konfigurowanie Sphinksa	250
Pliki utworzone przez program Sphinx	252
Tworzenie dokumentacji	252
Pisanie dokumentacji	253
Dyrektywy	254
Organizowanie dokumentów	254
Przykłady kodu	255
Generowanie dokumentacji na podstawie ciągów docstring	255
Testy dokumentacji	256
Konfigurowanie Sphinksa	257
Wpisy Todo	257
Budowanie warunkowe	258
Zmiana wyglądu i wrażenia	259
Jak napisać dobrą dokumentację?	259
Sekcje tekstu w dokumentacji technicznej	259
Inne narzędzia do tworzenia dokumentacji	261
MkDocs	261
Notatniki Jupyter	262
GitBook	262
Read the Docs	262
pydoc	262
S5	262
pygments	262
doctest	263
PyPDF2	263
pandoc	263
Najlepsze praktyki	263
Skorowidz	265

ROZDZIAŁ 2



Wyjątki w Pythonie

*Kiedys przypadkowo wstawiliśmy dodatkowe zero do nagłówka „Zostało zniszczonych 2000 czołgów”.
Władze były wściekle.*

— wspomnienia mojego dziadka z pracy w branży drukarskiej w 1941 roku

Jeśli program składa się tylko z jednego wiersza kodu, to w tym wierszu może być defekt i prędzej czy później defekt się znajdzie. O ile **defekty** polegają na tym, że kod nie robi tego, co powinien, o tyle **debugowanie** polega na ich usuwaniu. Jest to bardziej skomplikowane, niż się wydaje. Debugowanie oznacza kilka rzeczy:

- Wiemy, co powinien robić program.
- Wiemy, że w programie jest defekt.
- Uznajemy, że defekty muszą zostać usunięte.
- Wiemy, jak je usunąć.

W przypadku wielu subtelnych błędów pierwsze trzy punkty nie są banalne. Jednak gdy w programie napisanym w Pythonie wystąpi wyjątek (ang. *exception*), sytuacja jest dość oczywista: *chcemy, aby wyjątek zniknął*. Z tego powodu w tym rozdziale skoncentrujemy się na ostatnim punkcie: *jak usunąć defekt, o którego istnieniu wiemy*. Innymi problemami będziemy się zajmować w kolejnych rozdziałach.

Wyjątki są defektami, o których istnieniu wiemy

Niewiele programów działa płynnie od pierwszej próby. Zazwyczaj zanim program zacznie działać, zobaczymy co najmniej jeden komunikat o błędzie. Kiedy zobaczymy taki komunikat lub **wyjątek** w Pythonie, to będziemy wiedzieć, że coś jest nie tak z naszym kodem. Aby ponownie użyć metafory z rozdziału 1.: gdyby nasz program był budynkiem, to wyjątek oznaczałby, że dom się pali (rysunek 2.1). Ponieważ wyjątki w Pythonie zwykle występują z powodu defektów, nie ma wątpliwości co do tego, czy wyjątek jest błędem, czy nim nie jest. W związku z tym tego rodzaju defekty są stosunkowo łatwe do zdebugowania.

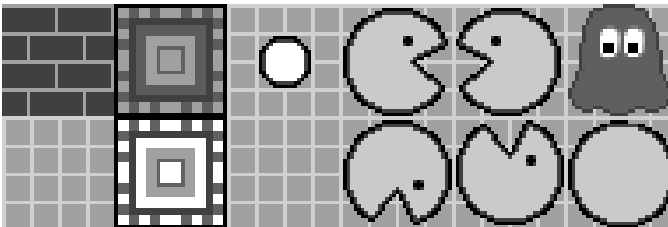


Wyjątki

Rysunek 2.1. Jeśli program jest budynkiem, to wyjątek występuje wtedy, gdy w budynku mamy pożar. Uciekanie od wyjątków nie ma sensu. Przynajmniej wtedy, gdy wiemy, że one są

W ramach przykładu przygotowujemy grafiki do gry MazeRun. Do utworzenia obrazu z graficznych płytek wykorzystamy bibliotekę Pygame. Płytki mają rozmiar 32×32 piksele. Można je ze sobą starannie połączyć, aby utworzyć poziomy i poruszające się obiekty. Wszystkie pliki znajdują się w pliku graficznym podobnym do pokazanego na rysunku 2.2. Musimy przeczytać plik obrazu i zapisać wszystkie kwadratowe płytki w słowniku Pythona, tak aby mieć do nich łatwy dostęp — na przykład wykorzystując znaki w roli kluczy:

```
tiles = {
    '#': wall_tile_object,
    ' ': floor_tile_object,
    '*': player_object,
}
```



Rysunek 2.2. Płytki, których będziemy używać do tworzenia grafiki w grze MazeRun. Chcemy utworzyć słownik, z którego będzie dostęp do płytek za pośrednictwem pojedynczych znaków (np. płytka ścienna w lewym górnym rogu będzie reprezentowana za pomocą znaku #). Sam obraz jest w formacie XPM (tiles.xpm). Format ten umożliwia łatwą obsługę przezroczystości w kodzie Pygame. Równie dobrze można jednak wykorzystać inne formaty

Podczas pisania kodu do utworzenia słownika zobaczymy typowe wyjątki spotykane w Pythonie. W tym rozdziale omówimy trzy proste strategie debugowania wyjątków:

1. Czytanie kodu w lokalizacji błędu.
2. Zrozumienie komunikatu o błędzie.
3. Przechwytywanie wyjątków.

W czasie wykonywania tych działań mamy nadzieję nauczyć się czegoś o ogólnej naturze defektów.

Czytanie kodu

Ogólnie rzecz biorąc, wyjątki w Pythonie dzielą się na dwie kategorie: wyjątki zgłaszane *przed uruchomieniem* kodu (SyntaxError) oraz wyjątki wywoływane *podczas działania kodu* (wszystkie inne). Python zaczyna interpretowanie i wykonywanie kodu wiersz po wierszu tylko wtedy, gdy nie znajdzie żadnych błędów typu SyntaxError. Od tego momentu mogą występować inne typy wyjątków.

Błędy typu SyntaxError

Błędy typu SyntaxError należą do tych wyjątków Pythona, które są najłatwiejsze do naprawienia. Ich podtypem są wyjątki typu IndentationError. W obu przypadkach Python nie interpretuje lub nie *tokenizuje* polecenia poprawnie, ponieważ zostało ono nieprawidłowo napisane. Tokenizacja jest wykonywana przed uruchomieniem jakiegokolwiek kodu. W związku z tym błędy SyntaxError zawsze pojawiają się w pierwszej kolejności. Przyczyną powstawania błędów SyntaxError zazwyczaj są różnego rodzaju literówki: brakujące znaki, znaki dodatkowe, znaki specjalne w nieprawidłowych miejscach itp. Spójrzmy na przykład. Przygotowywanie zestawu kafelków zaczniemy od zaimportowania biblioteki pygame:

```
import pygame
```

Wykonanie tej instrukcji się nie powiedzie. Zostanie przy tym wyświetlony irytujący komunikat, który widziałem pierwszego dnia mojej przygody z programowaniem i który prawdopodobnie zobaczę mojego ostatniego dnia w zawodzie programisty:

```
File "load_tiles.py", line 2
    import pygame
    ^
```

```
SyntaxError: invalid syntax
```

W instrukcji znalazło się nieprawidłowo napisane polecenie import, którego Python nie rozumie. Również Tobie prawdopodobnie przytrafiło się coś podobnego. W tym przypadku wystarczy odczytać kod w komunikacie o błędzie i zobaczyć, w czym jest problem. Czy poprzez czytanie kodu wskazanego w komunikacie o błędzie *zawsze* możemy ustalić, na czym polega wada? Aby się tego dowiedzieć, przyjrzymy się innym wyjątkom. Drugi powszechnie spotykany błąd typu SyntaxError często jest spowodowany brakującymi nawiasami. Przypuśćmy, że staramy się zdefiniować listę kafelków wraz z odpowiadającymi im indeksami x i y na obrazie:

```
TILE_POSITIONS =[
    ('#', 0, 0), # ściana
    (' ', 0, 1), # podłoga
    ('.', 2, 0), # punkt
    ('*', 3, 0), # gracz
```

Ten kod ulega awarii natychmiast po próbie jego uruchomienia w Pythonie:

```
SyntaxError: unexpected EOF while parsing
```

W komunikacie o błędzie Python nie udzielił nam odpowiedzi, że brakuje zamykającego nawiasu kwadratowego. Doszedł do końca pliku i zakończył działanie. Gdybyśmy jednak dodali nową linię na końcu pliku, na przykład rozmiar kafelków w pikselach:

```
SIZE = 32
```

to komunikat o błędzie zmieni się na:

```
File "load_tiles.py", line 11
    SIZE = 32
    ^
```

```
SyntaxError: invalid syntax
```

Zwróćmy uwagę, że w komunikacie numer wiersza kodu znalazł się *za* listą — jest to numer tego wiersza, który nie ma nic wspólnego z brakującym nawiasem. Po prostu nawias znalazł się tam, gdzie go nie oczekiwano. Programista Pythona musi szybko nauczyć się rozpoznawać objawy brakujących nawiasów. Dobre edytory tekstu liczą za nas nawiasy i delikatnie informują, gdy jakiegoś brakuje. Na podstawie komunikatu o błędzie `SyntaxError` można zidentyfikować defekt, ale opisy często są niedokładne. Bardziej niepokojącym aspektem brakujących nawiasów jest fakt, że w sytuacji, gdybyśmy zapomnieli o nawiasie otwierającym, zobaczylibyśmy zupełnie inny typ wyjątku:

```
TILE_POSITIONS = ('#', 0, 0), # ściana
                 (' ', 0, 1), # podłoga
                 ('.', 2, 0), # punkt
                 ('*', 3, 0), # gracz
                 ]
```

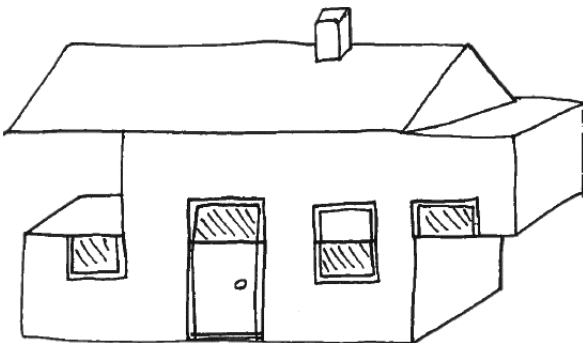
Próba interpretacji tego wiersza spowoduje wyświetlenie następującego komunikatu o błędzie:

```
IndentationError: unexpected indent
```

Python nie wie, dlaczego w powyższym kodzie znajduje się wcięcie w drugim wierszu. Zwróćmy uwagę, że wyjątek `IndentationError` powstaje tylko wtedy, gdy pierwszy element listy jest podawany w wierszu, w którym znajduje się instrukcja przypisania. W przeciwnym razie ponownie uzyskamy błąd `SyntaxError`. Tego rodzaju błędy są bardzo częste, ale zazwyczaj najłatwiejsze do naprawy.

■ **Wniosek** Podobne wady kodu Pythona mogą prowadzić do różnych komunikatów o błędach.

Innym powodem, obok brakującego nawiasu, powstawania wyjątku `IndentationError` są nieodpowiednie wcięcia. Ze złymi wcięciami mamy do czynienia wtedy, gdy wskażemy nowy blok kodu za pomocą dwukropka (:), ale zapomnimy o wcięciu. Błędne wcięcie występuje w przypadku, gdy użyjemy jednego odstępu więcej lub mniej w stosunku do wiersza poprzedzającego. Najgorszy przypadek złych wcięć występuje wtedy, kiedy użyjemy tabulacji zamiast spacji w którymś miejscu w pliku. Takie błędy są bowiem trudne do zauważenia. Można tego uniknąć dzięki korzystaniu z edytora przeznaczonych do pisania kodu w Pythonie. Na szczęście objawy złych wcięć są często oczywiste (patrz rysunek 2.3), a lokalizację wyjątku `IndentationError` możemy znaleźć na podstawie numeru wiersza w komunikacie o błędzie.



Rysunek 2.3. Wyjątek `IndentationError` w przypadku, gdyby programy były budynkami

Najlepsze praktyki debugowania wyjątków `SyntaxError`

Wyjątek `SyntaxError` lub jego podtyp `IndentationError` często można rozwiązać przez uważne czytanie wiersza kodu wskazanego w komunikacie o błędzie. Strategia jest nieco podobna do słynnego *veni, vidi, vici* Juliusza Cezara: najpierw *przechodzimy* do wiersza wskazanego w kodzie (*veni*), następnie *oglądamy* kod w tej lokalizacji (*vidi*) i na koniec usuwamy problem (*vici*). Stosowanie tej strategii w praktyce pozwala rozwiązać wiele wyjątków Pythona w bardzo krótkim czasie. Oto najbardziej typowe poprawki błędów `SyntaxError`:

- Najpierw przyjrzyj się wierszowi podanemu w komunikacie o błędzie.
- Przyjrzyj się wierszowi znajdującemu się bezpośrednio nad wierszem wymienionym w komunikacie.
- Wytnij i wklej blok kodu z błędem do osobnego pliku. Czy wyjątek `SyntaxError` nadal występuje w pozostałym kodzie? (Oczywiście mogą być inne błędy).
- Sprawdź, czy nie brakuje dwukropków po poleceniach takich jak `if`, `for`, `def` lub `class`.
- Sprawdź, czy nie brakuje nawiasów. Łatwiej je znaleźć za pomocą dobrego edytora.
- Sprawdź, czy w kodzie nie ma otwartych cudzysłowów — zwłaszcza w tekstach obejmujących wiele wierszy.
- Ujmij w komentarz wiersz wskazany w komunikacie o błędzie. Czy błąd się zmienił?
- Sprawdź wersję Pythona, którą się posługujesz (czy używasz polecenia `print` bez nawiasów kwadratowych w Pythonie 3?).
- Użyj edytora, który wstawia cztery spacje za każdym razem, gdy użytkownik naciśnie klawisz `Tab`.
- Zadbaj o to, aby Twój kod był zgodny z zasadami PEP8 (patrz rozdział 14.).

Analiza komunikatów o błędach

W poprzednim punkcie użyliśmy strategii *veni, vidi, vici* do naprawiania błędów `SyntaxError`. Ogólnie rzecz biorąc, polegała ona na dokładnym przyglądaniu się wierszowi wymienionemu w komunikacie o błędzie. Czy ta strategia sprawdzi się dla wszystkich błędów? Biorąc pod uwagę, że mamy przed sobą pięć dodatkowych rozdziałów o debugowaniu, to chyba nie. Przyjrzyjmy się nieco bardziej skomplikowanemu przykładowi. Aby utworzyć obraz, utworzymy słownik w celu wyszukiwania prostokątów płytek do skopiowania. Te prostokąty są obiektami `pygame.Rect`. Prostokąty tworzymy w pomocniczej funkcji `get_tile_rect()`, a słownik kafelków — w funkcji `load_tiles()`. Oto pierwsza implementacja:

```
from pygame import image, Rect, Surface

def get_tile_rect(x, y):
    """Konwertuje indeksy kafelków na obiekty pygame.Rect"""
    return Rect(x * SIZE, y * SIZE, SIZE, SIZE)

def load_tiles():
    """Zwraca słownik prostokątów z kafelkami"""
    tiles = {}
    for symbol, x, y in TILE_POSITIONS:
        tiles[x] = get_tile_rect(x, y)
    return tiles
```

Możemy teraz wywołać funkcję i spróbować wyodrębnić ze słownika kafelek ściany (w skrócie `'#'`):

```
tiles = load_tiles()
r = get_tile_rect(0, 0)
wall = tiles['#']
```

Próba uruchomienia tego kodu powoduje jednak błąd `KeyError`:

```
Traceback (most recent call last):
  File "load_tiles.py", line 32, in <module>
    wall = tiles['#']
KeyError: '#'
```

Niezależnie od tego, jak bardzo wpatrujemy się w linię 32., nie znajdujemy niczego złego w żądaniu '#' ze słownika płytek. Właśnie w taki sposób powinien działać nasz słownik. A jeśli błędu nie ma w linii 32., to logicznie możemy wywnioskować, że *musi być gdzieś indziej*.

■ **Wniosek** Lokalizacja podana w komunikacie o błędzie nie zawsze jest lokalizacją defektu.

Jak można znaleźć defekt? Aby uzyskać więcej informacji, przyjrzyjmy się bliżej komunikatowi o błędzie. Czytanie komunikatów o błędzie generowanych przez Pythona nie jest zbyt trudne. Komunikat taki w Pythonie zawiera trzy istotne informacje: **typ błędu**, **opis błędu** i tzw. **ślad** (ang. *traceback*). Zajmijmy się nimi.

Typ błędu

Z technicznego punktu widzenia komunikat o błędzie oznacza, że w Pythonie powstał wyjątek. Typ błędu wskazuje na klasę zgłoszonego wyjątku. Wszystkie wyjątki są podklasami klasy `Exception`. W Pythonie 3.5 występuje łącznie 47 różnych typów wyjątków. Ich pełną listę można wyświetlić za pomocą następującego polecenia:

```
[x for x in dir(__builtins__) if 'Error' in x]
```

Hierarchiczne relacje pomiędzy tymi klasami przedstawiono na schemacie na rysunku 2.4. Jak można zauważyć, wiele typów błędów jest związanych z wejściem-wyjściem. Intrygujące jest również to, że istnieją cztery odrębne kategorie związane z `Unicode`.

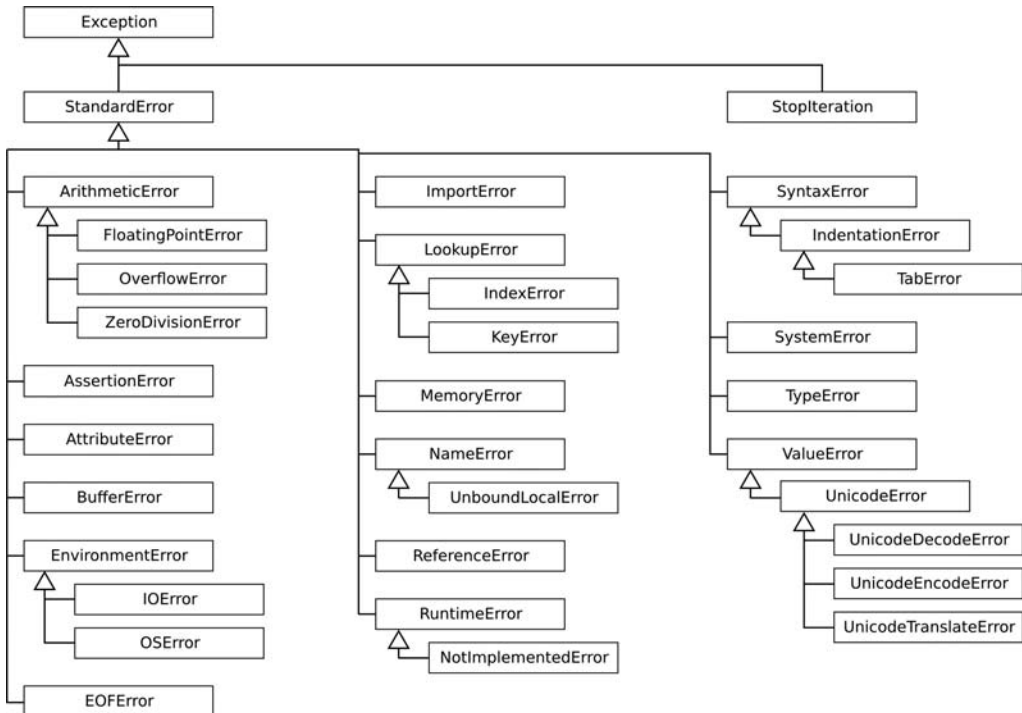
Od wydania Pythona 3 i wprowadzenia *Unicode* liczba możliwości popełnienia błędów w pisowni znaków zwiększyła się o kilka rzędów wielkości. Znajomość możliwych typów wyjątków oraz ich znaczenia to solidna wiedza podstawowa dla doświadczonych programistów Pythona. W naszym przypadku wystąpienie wyjątku `KeyError` daje czytelną wskazówkę, że staraliśmy się wyszukać w słowniku coś, czego tam nie było.

Opis błędu

Tekst występujący bezpośrednio za typem błędu dostarcza opisu tego, na czym dokładnie polegał problem. Opisy w komunikatach o błędach są czasami bardzo dokładne, a czasami nie. Na przykład w przypadku wywoływania funkcji ze zbyt dużą lub zbyt małą liczbą argumentów w komunikacie o błędzie uzyskamy dokładne liczby:

```
TypeError: get_tile_rect() takes 2 positional arguments but 3 were given
```

To samo dotyczy niepowodzeń w rozpakowywaniu krotek. W innych przypadkach Python grzecznie mówi nam, że nie ma pojęcia, co poszło nie tak. Do tej grupy należy większość wyjątków typu `NameError`. W przypadku naszego `KeyError` jedyną informacją, którą otrzymujemy, jest znak '#'. Wewnętrzny głos doświadczonego programisty szybko automatycznie uzupełnia tę informację do takiej oto postaci:



Rysunek 2.4. Hierarchia wyjątków w Pythonie. Na ilustracji pokazano dziedziczenie dla 34 spośród 47 wyjątków w Pythonie 3.5. Bardzo szczegółowe typy, w większości podklasy klasy `IOError`, pominięto dla poprawienia czytelności rysunku

Drogi Użytkowniku,

dziękuję za Twoje ostatnie polecenie. Starałem się wydobyć wartość '#' ze słownika `tiles`, tak jak poleciłeś. Ale mimo przejrzania słownika nie mogłem jej w nim znaleźć. Szukałem wszędzie i jej tam nie ma. Czy na pewno nie umieściłeś wpisu gdzieś indziej? Naprawdę mi przykro. Mam nadzieję, że następnym razem pójdzie lepiej.

Na zawsze Twój, Python

Ślad

Ślad zawiera szczegółowe informacje o tym, *gdzie* w kodzie powstał wyjątek. Ślad zawiera następujące elementy:

1. Kopię uruchamianego kodu. Czasami udaje się dostrzec w nim błąd natychmiast. Nie tym razem.
2. Numer wiersza, który był uruchamiany w chwili, gdy wystąpił błąd. Defekt **musi** być dokładnie w tej linijce albo w linijce, która była uruchamiana wcześniej.
3. Wywołania funkcji, które doprowadziły do błędu. Ślad można czytać tak, jakby był łańcuchem zdarzeń: *Moduł wywołał funkcję X, która wywołała funkcję Y, która z kolei uległa awarii, zgłaszając wyjątek*. Oba zdarzenia wyświetlają się w odrębnych wierszach śladu. Czytanie dłuższych śladów **warto zacząć od dołu**. Nie oznacza to, że informacja o przyczynie błędu jest zawsze na końcu. Dość często jednak informacje na końcu śladu dostarczają podpowiedzi, gdzie należy szukać problemu.

Dedukcja

Szukając przyczyn wyjątków `KeyError`, możemy stosować **dedukcję**: jeśli nie było klucza '#' w słowniku, to czy w ogóle ten klucz został zapisany? W którym wierszu został zapisany klucz? Czy interpreter dotarł do tego wiersza? Jest kilka miejsc w kodzie, gdzie przepływ danych do słownika `tiles` mógł zostać przerwany. Podczas analizy funkcji `load_tiles` można zauważyć przypisanie nieprawidłowych kluczy. Polecenie przypisania to:

```
tiles[x] = get_tile_rect(x, y)
```

podczas gdy powinno być:

```
tiles[symbol] = get_tile_rect(x, y)
```

Przeczytanie i zrozumienie komunikatu o błędzie okazało się przydatne do zidentyfikowania usterki. Nawet jeśli defekt jest bardziej złożony, to komunikat o błędzie zazwyczaj daje nam punkt wyjścia — informację o tym, gdzie należy szukać źródła problemu. Zauważyliśmy jednak, że trochę dedukcji było konieczne, ponieważ wyjątek wystąpił w innym wierszu niż ten, gdzie była wada. Istnieje wiele możliwych wad, które mogłyby doprowadzić do tego samego objawu. Czasami trzeba sprawdzić kilka lokalizacji. W krótkim fragmencie kodu możemy zastosować dedukcję intuicyjnie i sprawdzić wiele potencjalnych lokalizacji defektów. Bardziej systematyczne podejście zaprezentujemy w rozdziale 4.

Przechwytywanie wyjątków

Gdy instrukcje importu i słownik `tiles` działają prawidłowo, możemy spróbować załadować obraz z kafelkami:

```
from pygame import image, Rect
```

```
tile_file = open('tileless.xpm', 'rb')
```

Powyższa próba nie powiedzie się z następującym, niezbyt jasnym komunikatem o błędzie:

```
FileNotFoundError: [Errno 2] No such file or directory: 'tileless.xpm'
```

Błąd polega na literówce w nazwie pliku. `FileNotFoundError` jest podklasą klasy `IOError`. „Rodzeństwo” klasy `FileNotFoundError` to bardzo popularne błędy przetwarzania danych. W niektórych projektach połowę moich błędów stanowiły wyjątki `IOError`. Na szczęście to jest precyzyjny komunikat, który pozostawia niewiele miejsca na interpretację. Ten błąd można naprawić przez uważne sprawdzanie ścieżki i nazwy pliku w kodzie. Aby naprawić błąd, musimy dowiedzieć się, gdzie naprawdę jest plik, a następnie ponownie sprawdzić pisownię nazwy pliku w kodzie. Czasami potrzebnych jest kilka prób ze względu na kłopotliwe szczegóły: ścieżki względne i bezwzględne, brakujące znaki podkreślenia, myślniki oraz, co nie umniejsza ich rangi, znaki lewego ukośnika (ang. *backslash*) w systemie Windows, które w ciągach znaków Pythona powinny być pisane jako podwójne lewe ukośniki (`\\`). **Wady kodu powodujące zgłoszenie wyjątku `IOError` polegają prawie zawsze na błędnym podaniu nazwy pliku.**

To jasne, że nie możemy zapobiec wszystkim wyjątkom w Pythonie. Co jeszcze możemy zrobić? Jedną z możliwości jest **reagowanie na wyjątki** wewnątrz programu. Staramy się wykonać operację, zdając sobie sprawę, że może się nie powieść. Jeśli się nie powiedzie, Python zgłosi wyjątek. Konstrukcja `try.. except` pozwala na zdefiniowanie odpowiedniej reakcji. Typową sytuacją, w której może się przydać przechwytywanie wyjątków, są nazwy plików wprowadzane przez użytkownika:

```
filename = input("Wprowadź nazwę pliku: ")
try:
    tiles = load_tile_file(filename)
except IOError:
    print("Nie znaleziono pliku: {}".format(filename))
```

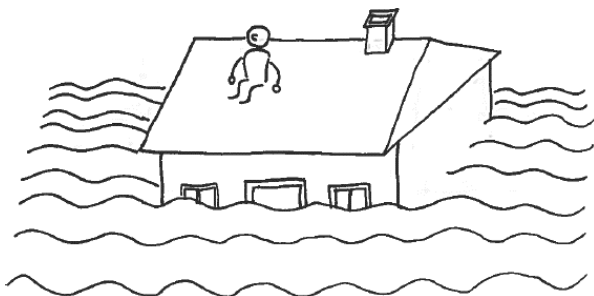
Instrukcja `except` umożliwia zdefiniowanie właściwej reakcji na specyficzne typy wyjątków. Strategię tę nazwano **EAFP** (od ang. *Easier to Ask Forgiveness than Permission* — dosł. łatwiej poprosić o wybaczenie niż o zgodę). *Prośba o przebaczenie* oznacza zareagowanie na wyjątek, *prośba o zgodę* oznacza sprawdzenie, czy plik istnieje przed próbą jego otwarcia. Takie rozwiązanie jest łatwiejsze, ponieważ sprawdzanie z góry wszystkich możliwości tego, co może pójść źle, nie jest ani realne, ani pożądane. Programista Pythona *Alex Martelli* stwierdził, że przechwytywanie wyjątków jest świetną strategią dla reagowania na nieprawidłowe dane wejściowe lub ustawiania konfiguracji oraz ukrywania wyjątków przed użytkownikami. Przechwytywanie wyjątków jest również przydatne do zapisywania ważnych danych przed zakończeniem programu. Jednak technika przechwytywania wyjątków ma także wielu przeciwników. Joel Spolsky, dobrze znany autorytet w tworzeniu oprogramowania, stwierdza:

Uważam przechwytywanie wyjątków za technikę nieróżniącą się od stosowania instrukcji „goto” uznawanych od lat sześćdziesiątych ubiegłego wieku za szkodliwe ze względu na to, że powodują nagły skok z jednego punktu kodu do innego.

W rzeczywistości ścieżka, którą wyjątki „poruszają się” w programie, jest *niewidoczna* w kodzie. Patrząc na funkcję w Pythonie, nie widzimy, jakie wyjątki mogą być zgłoszone wewnątrz lub czy powstałe wyjątki doprowadzą do zatrzymania programu. W związku z tym uwzględnienie wszystkich możliwych ścieżek działania staje się bardzo trudne, a to sprawia, że bardzo łatwo może dojść do wprowadzenia dodatkowych błędów. Musimy również być bardzo ostrożni w podejmowaniu decyzji o tym, *jakie* wyjątki należy przechwytywać. Na pewno użycie konstrukcji `try... except` w postaci zaprezentowanej poniżej jest bardzo złym pomysłem:

```
try:
    wywołania_jakichś_funkcji()
except:
    pass
```

Taką konstrukcję określa się jako *wzorzec pieluchy*. Przechwytuje wszystko, ale po pewnym czasie nie chcesz zaglądać do środka. Dzięki stosowaniu tej konstrukcji wyjątki znikają, ale w zamian powstaje gorszy problem: wyjątki są „przykryte”, ale „przykryte” są również nasze możliwości zdiagnozowania tego, co się dzieje (patrz rysunek 2.5). Najlepsza praktyka polega na używaniu konstrukcji `try... except` wyłącznie w ściśle określonych sytuacjach i zawsze w celu przechwycenia konkretnego typu wyjątku.



Rysunek 2.5. Gdyby wyjątki były pożarem w budynku, to tak wyglądałaby instrukcja `except: pass`. Z pewnością ugasimy wyjątek, ale czy naprawę tego chcielibyśmy?

Najlepsze praktyki debugowania wyjątków `IOError`

Ponieważ wyjątki `IOError` są częste i bardzo denerwujące dla początkujących, nie zaszkodzi wyliczyć najbardziej popularnych strategii przeciwdziałania tego rodzaju błędom:

- Znajdź dokładną lokalizację pliku w terminalu lub przeglądarce plików.
- Wyświetl ścieżkę i nazwę pliku używane w programie. Porównaj je z rzeczywistymi.

- Sprawdź bieżący katalog roboczy (`import os; print(os.getcwd ())`).
- Zastąp ścieżki względne bezwzględnymi.
- **W systemie Unix:** sprawdź, czy masz uprawnienia dostępu do określonego pliku.
- Skorzystaj z modułu `os.path` do obsługi ścieżek i katalogów.
- Uważaj na lewe ukośniki w ścieżce! Aby uzyskać poprawny separator, należy zastąpić je ukośnikami zwykłymi (/) lub podwójnymi lewymi ukośnikami (\\).

Błędy i defekty

W tym rozdziale widzieliśmy trzy strategie obsługi **defektów**, które powodowały wyjątki: debugowanie poprzez przeglądanie kodu, debugowanie poprzez przeglądanie komunikatu o błędzie oraz przechwytywanie wyjątku. Spróbujmy podsumować nasze obserwacje: czasami komunikat o błędzie wskazuje bezpośrednio na defekt (np. `SyntaxError`). Istnieje również wiele przykładów, kiedy komunikat o błędzie jest wystarczająco precyzyjny do tego, by zawęzić defekt do kilku możliwości (np. `IOError`). Inne komunikaty o błędach są bardziej niejasne, a ustalenie, co oznaczają, wymaga doświadczenia. Do tego celu przydatne mogą się okazać strategie przeciwdziałania błędom zamieszczone w tym rozdziale. Pokazaliśmy jednak również, że defekt może być bardzo daleko od lokalizacji wskazanej w komunikacie o błędzie. W takiej sytuacji komunikat o błędzie nie jest zbyt pomocny. Dostarcza wskazówki jak wyrocznia, ale niemożliwe jest zlokalizowanie usterki poprzez czytanie samego komunikatu o błędzie. W Pythonie zwykle istnieje wiele potencjalnych defektów, które mogą prowadzić do tego samego błędu. Ta sytuacja jest bardzo częsta dla wyjątków `TypeError`, `ValueError`, `AttributeError` i `IndexError`. Inna trudna sytuacja występuje w przypadku, gdy dostarczymy niewłaściwych danych do funkcji bibliotecznej, na przykład `pygame.Rect`. W rezultacie powstanie wyjątek w bibliotece, pomimo że defekt jest w naszym kodzie. W takich sytuacjach trzeba uwzględnić wszystkie informacje z komunikatu o błędzie: **miejsce w kodzie, typ błędu i ślad**. Istnieje wiele przypadków, kiedy te informacje są wystarczające do zlokalizowania defektu, dzięki czemu opisany sposób jest dobrą, *intuicyjną* strategią debugowania.

A co z wyciszeniem wyjątków za pomocą konstrukcji `try . . except`? Przechwytywanie wyjątków to świetna strategia radzenia sobie z sytuacjami *wyjatkowymi*, które są poza naszą kontrolą: na przykład nieprawidłowymi danymi wejściowymi lub nieprawidłowymi nazwami plików. Jednak obsługa wyjątków jest niewystarczająca do usunięcia defektów, które już znajdują się *wewnątrz* naszego programu. Program nie będzie działał lepiej tylko dzięki udawaniu, że wszystko jest w porządku. Konstrukcje `try . . except` pokazują, że możemy zarządzać błędami zgłaszanymi przez program nawet wtedy, gdy nie jesteśmy świadomi wad, które je powodują. Wynika stąd wniosek, który przeanalizujemy dokładniej w następnym rozdziale: **błędy i defekty to dwie różne rzeczy**. Błąd jest czymś, co obserwujemy — objawem, że coś poszło źle. Z kolei defekt jest ukryty gdzieś w kodzie. Aby naprawić kod, najpierw trzeba znaleźć defekt, który go powoduje. Wyszukiwanie defektów staje się **problemem dedukcji**.

Skąd się biorą defekty?

Dlaczego w ogóle wprowadzamy pojęcie defektów? Powody występowania defektów w programach są różnorodne. Aby skutecznie debugować, warto wiedzieć, skąd pochodzą defekty. Oto w jaki sposób popełniałem większość błędów w moich programach w Pythonie:

- Po pierwsze, **błędy występowały w trakcie implementacji**. Kod był w mojej głowie, ale coś poszło nie tak po drodze do edytora tekstu: brakujący dwukropki, błąd w pisowni zmiennej, zapomniany parametr. Albo zapomniałem, w jaki sposób używa się funkcji, i dodałem nieprawidłowy parametr. Większość tych defektów szybko doprowadzi do awarii, często będzie się wiązała ze zgłoszeniem wyjątku.
- Po drugie, **złe planowanie** prowadziło do bardziej subtelnych defektów. Kod był niepoprawny już w mojej głowie: wybrałem niewłaściwe podejście, zapominałem o bardzo ważnym szczególe, co skończyło się tym, że rozwiązywałem inny problem, niż początkowo chciałem. Tego rodzaju defekty są trudniejsze do rozpoznania. Zwykle skutkują tym, że muszę rozpocząć pisanie całego fragmentu kodu od początku. Dobrą strategią pozwalającą na wczesne wykrycie złego planowania jest **testowanie**.

- Po trzecie, pośrednio do defektów przyczyniał się **zły projekt**. Zawsze jeśli pisałem nadmiarowy kod i przywiązywałem niewiele wagi do porządkowania kodu lub nie dokumentowałem tego, co robię, to *późniejsze modyfikacje* programu z większym prawdopodobieństwem przyczyniały się do powstawania błędów. Aby uniknąć tego rodzaju problemów, potrzebujemy najlepszych praktyk **utrzymywania projektu oprogramowania**.
- Wreszcie były **czynniki ludzkie**. Podczas korzystania z funkcji języka lub biblioteki po raz pierwszy, kiedy komunikacja z innymi programistami była trudna oraz w przypadku pisania programów w pośpiechu lub w zmęczeniu — defekty się mnożyły. Oprócz praktyk wymienionych wcześniej bardzo pomocną jest odpowiednia ocena własnych możliwości. Nie ufaj ślepo własnemu kodowi, ponieważ będzie on zawierać defekty częściej niż tylko od czasu do czasu.

Python nie jest najłatwiejszym językiem do debugowania. Dynamiczne typowanie zmiennych w Pythonie powoduje bardzo ogólne komunikaty o błędach, które wymagają uważnej interpretacji. W innych językach kompilator dostarcza bardziej precyzyjnych komunikatów, które pomagają w tworzeniu kodu *wykonywalnego*. Z drugiej strony Python daje wiele możliwości bliskiej interakcji z kodem. Umożliwia analizowanie defektów z bliskiej odległości. Ta cecha pozwala *gasić pożary po kolei* i eliminować wyjątki natychmiast po ich powstaniu. Jeśli chcemy, żeby nasze programy działały poprawnie, musimy wykorzystać tę mocną stronę Pythona. Ponieważ samo patrzeć na komunikat o błędzie nie jest wystarczające, to debugowanie lub nawet odnajdowanie bardziej wymagających defektów wymaga zapoznania się z innymi technikami. Zanim przejdziemy do metod systematycznego debugowania, w kolejnym rozdziale dokładniej przeanalizujemy naturę defektów.

Poprawny kod

Przed przejściem do następnego rozdziału warto dokończyć kod odpowiedzialny za ładowanie kafelków. Po zdebugowaniu instrukcji importu, listy kafelków i funkcji `load_tiles` możemy dodać kilka wierszy odpowiedzialnych za kompozycję obrazu złożonego z trzech kafelków. Poniżej zamieszczono kompletny kod:

```
from pygame import image, Rect, Surface

TILE_POSITIONS = [
    ('#', 0, 0), # ściana
    (' ', 0, 1), # podłoga
    ('.', 2, 0), # punkt
    ('*', 3, 0), # gracz
]

SIZE = 32

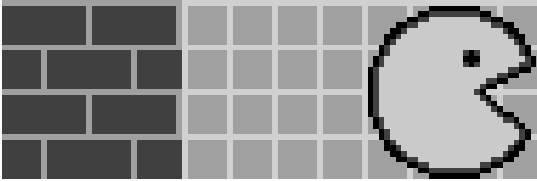
def get_tile_rect(x, y):
    """Konwertuje indeks kafelka na obiekt pygame.Rect"""
    return Rect(x * SIZE, y * SIZE, SIZE, SIZE)

def load_tiles():
    """Zwraca słownik prostokątów kafelków"""
    tile_image = image.load('tiles.xpm')
    tiles = {}
    for symbol, x, y in TILE_POSITIONS:
        tiles[symbol] = get_tile_rect(x, y)
    return tile_image, tiles

if __name__ == '__main__':
    tile_img, tiles = load_tiles()
    m = Surface((96, 32))
```

```
m.blit(tile_img, get_tile_rect(0, 0), tiles['#'])
m.blit(tile_img, get_tile_rect(1, 0), tiles[' '])
m.blit(tile_img, get_tile_rect(2, 0), tiles['*'])
image.save(m, 'tile_combo.png')
```

Uruchomienie kodu powoduje wyświetlenie obrazu, który możemy przyjąć za dowód, że kompozycja większych grafik z kafelków jest możliwa (patrz rysunek 2.6). Być może przykład z kafelkami już zachęcił Cię do podjęcia samodzielnych prób uruchamiania kodu. Wkrótce będziesz mieć wiele możliwości debugowania własnych komunikatów o błędach. Kod jest zapisany w pliku *maze run/load tiles.py* dostępnym pod adresem https://github.com/krother/maze_run.



Rysunek 2.6. Pomyślnie skomponowane kafelki

Najlepsze praktyki

- Zły kod, który prowadzi do błędu, nazywa się **defektem**.
- Z niektórymi wyjątkami wiążą się precyzyjne komunikaty o błędach. Takie błędy można poprawić przez analizowanie kodu.
- Niektóre defekty można znaleźć w lokalizacjach wskazanych przez komunikat o błędzie.
- Inne defekty są daleko od lokalizacji określonej w komunikacie o błędzie.
- Komunikaty o błędach zawierają typ błędu, opis i ślad.
- Strategią zmierzającą do ustalenia przyczyn błędu jest **dedukcja**.
- **Przechwytywanie wyjątków** za pomocą konstrukcji `try... except` to strategia radzenia sobie ze specyficznymi sytuacjami i typami błędów.
- Instrukcji `except` **zawsze** należy używać z określonym typem wyjątku.
- **Nigdy** nie należy używać instrukcji `except`, w której wnętrzu jest instrukcja `pass`.
- **Błędy i defekty** to dwie różne rzeczy.

Skorowidz

A

Agile, 25
Anaconda, 82
analiza
 eksploracyjna danych, 164
 wymagań, 219, 220
Anderson David, 233
asercja, 237, 238, 246
atrybut, 84, 87, 94
 add, 86
 file, 86
 getitem, 86
 name, 86

B

baza danych, 240, 241
 SQLite, 241
 zalety, 241
Beck Kent, 216
biblioteka
 asyncio, 166
 gevent, 166
 NumPy, 239, 246
 pandas, 240
 pomocnicza, 135, 166
 Pygame, 22, 59
 instalacja, 29, 202
 okno, 67
 pygments, 262
 PyPDF2, 263
 Twisted, 166

BitBucket, 188
blok except-try, 101
błąd, 33, 42, 44, 51, 62, 149, *Patrz*
 też: defekt, wyjątek
 automatyzowanie, 62
 komunikat, 33, 37, 38
 analiza, 37, 40
 nieodtworzalny, 62
 opis, 38
 powtarzalność, 61
 propagacja, 56, 57
 semantyczny, 45, 46, 47, 48,
 49, 50, 56, 57, 60, 72
 SyntaxError, 35, 36, 37
 śląd, 38, 39
 typ, 38
 UnboundLocalError, 48
breakpoint, *Patrz*: pułapka
Brooks Fred, 167

C

callback, *Patrz*: wywołanie
 zwrotne
ciąg
 dokumentacyjny, 81, 91, 94
 formatujący, 73
Codecept.js, 157
commit, 177, 178, 187, 188
cProfile, 111
Cucumber, 157
CVS, 188
Cython, 242, 243

D

dane
 analiza eksploracyjna, 164
 baza, *Patrz*: baza danych
 przypadek
 brzegowy, 123, 124, 164
 skrajny, *Patrz*: dane
 przypadek brzegowy
testowe, 129, 130, 164
 fikstura, *Patrz*: fikstura
 kolekcja, 139
 losowe, 138, 139
 moduł, 142
wejściowe, 47, 57, 71, 194
 puste, 156
 upraszczanie, 74, 75, 76
wyjściowe, 194
 złożone, 166
wynikowe, 137, 138
 oczekiwane, 47, 57, 133
debuger, *Patrz też*: debugowanie
 graficzny, 109
 interaktywny, 98
 ipdb, *Patrz*: ipdb
 pdb, *Patrz*: pdb
 webowy, 111
 wizualny, 110
debugowanie, 33, 42, 60, 61, 72,
 82, 167, *Patrz też*: debuger
 notatki, 67
 pomoc, 65, 66
 przez przeglądanie kodu, 42

debugowanie

- poprzez przeglądanie komunikatu o błędach, 42
- post mortem, 100, 101
- strategia, 34

defekt, 33, 42, 44, 45, 51, *Patrz też:*

błąd

- instrukcji przepływu sterowania, 53
- izolowanie, 63
- nakładanie, 57
- obsługa, 42
- propagacja, 56, 57, 60, 236
- równoważenie, 57
- śledzenie, 98
- wyszukiwanie
 - binarne, 64
 - dedukcja, 47, 57
 - metoda naukowa, 60, 61, 68
 - narzędzia, 71, 72
 - związany z funkcją, 55

dekompozycja zadań

- programowania, 219, 220
- lekka, 220
- podział na części, 221
- techniki, 220

dekorator

- @mock.patch, 135
- @pytest.fixture, 130, 131, 134
- @pytest.mark.parametrize, 132, 133
- funkcji, 90

Django, 87, 111, 165, 192, 196, 241

dług techniczny, 68

docstring, *Patrz:* ciąg

- dokumentacyjny
- doctest, 256, 257, 263

dokumentacja, 249

- EPUB, 253
- HTML, 252
- na podstawie docstring, 255
- PDF, 253
- sekcje tekstu, 259
- testowanie, *Patrz:* test
 - dokumentacji, doctest
 - tworzenie, 252, 253, 259

E

- EAFP, 41
- Easier to Ask Forgiveness than Permission, *Patrz:* EAFP
- edytor tekstu, 30
- Enthought Canopy, 82
- exception, *Patrz:* wyjątek

F

- fiktura, 130
 - baza danych, 139
 - parametryzowana, 133, 134
 - sprzątanie, 136
 - tmpdir, 137
- fixture, *Patrz:* fiktura
- format
 - .epub, 251
 - .rst, 250, 253
 - HTML, 252
- framework
 - Django, *Patrz:* Django
 - doctest, 126
 - nose, 126
 - py.test, 115, 117, 131, 134, 137, 142, 143, 144, 147, 148, 149, 193, 200
 - instalacja, 116
 - opcje, 151
 - sprzątanie po testach, 136
 - Selenium, *Patrz:* Selenium
 - testowy, 115, 126, 144, 165
 - unittest, 126, 144

funkcja

- %debug, 101
- %paste, 82, 83
- %run, 83
- %timeit, 158
- debug_print, 78
- dekorator, *Patrz:* dekorator
 - funkcji
- difflib.ndiff, 137
- exit, 85
- filecmp.cmp, 137
- getattr, 87
- hasattr, 87

introspekcji,

- Patrz:* introspekcja
- magiczna, 82
- pomocnicza, 119, 142
- pygame.event.poll, 59
- pygame.event.pump, 59
- pytest.raises, 122
- quit, 85
- range, 51
- setup_function, 136
- setup_module, 136
- sygnatura, 244
- teardown_function, 136
- teardown_module, 136
- testowa, 64
 - refaktoryzacja, 144
- wyodrębnianie, 212
- wywołania zwrotnego, *Patrz:* wywołanie zwrotne
- wywołanie, 55

G

generator

- labiryntów, 176
- liczb losowych, 61

GitBook, 262

gra MazeRun, *Patrz:* MazeRun
GUI, 165

H

hacking, 23, 24

haker, 23, 25

heisenbug, 62, 166

hermetyzacja, 90

hipoteza, 67, 72

alternatywna, 63

druga, 62

pierwsza, 62

przewidywanie, 62, 63

historijka użytkownika, 221

kryteria akceptacji, 222

opis przypadków użycia, 223

struktura, 221

wymagania, 222

I

idiom, 211, 213
indeks, 51, 52
instrukcja
 assert, 117, 118, 120, 121, 131,
 156, 237, 238, 239
 blit, 75, 77
 except, 41, 44
 if, 213
 pass, 44
 pprint, 74, 100
 print, 71, 72, 73, 79, 80
 w instrukcji if, 76
 przepływu sterowania
 defekt, *Patrz:* defekt
 instrukcji przepływu
 sterowania
 return, 55
interfejs
 programu, 213
 użytkownika graficzny,
 Patrz: GUI
introspekcja, 81, 84, 91, 93, 95, 98
 znajdowanie literówek, 94
inżynieria programowania, 25
ipdb, 98, 100
 instalowanie, 99
 konfigurowanie, 105
 uruchamianie, 99, 105
 w odpowiedzi na wyjątek, 101
 wiersz poleceń, 102, 103
ipdbplugin, 110
IPython, 82
 polecenie, *Patrz:* polecenie
 powłoka, 81, 84
 tryb
 Anaconda, *Patrz:* Anaconda
 Enthought Canopy, *Patrz:*
 Enthought Canopy
 samodzielnej konsoli, 82

J

język
 programowania
 C, 241
 C++, 241

Go, 166
 Scala, 166
 UML, 220
 Jupyter, 262

K

Kaizen, 168
Kanban, 168, 233
katalog, 83
 .hg, 194
 bin/, 194
 build/, 194
 danych, 194
 dist/, 194
 domowy, 193
 roboczy, 84
 sdist/, 194
klasa
 MagicMock, 135
 testu, 143, 144
klucz, 84
kod, *Patrz też:* program
 czytelność, 206, 217
 idiomatyczny, 211
 niezorganizowany, 204, 205,
 206, 208, *Patrz też:*
 oprogramowanie entropia
 ocena punktowa, 215
 porządkowanie, 203, 208, 209,
 210
 projekt, 208
 przegląd, 65, 66, 167
 redundancja, 206, 207
 refaktoryzacja, *Patrz:*
 refaktoryzacja
 restrukturyzacja, 207
 sprzątanie, 68
 standard
 PEP8, *Patrz:* standard PEP8
 struktura, 206, 217
 śmierdzący, 206
 unpythonic, 206
 uporządkowany, 214, 216, 217
 w języku C, 241, 246
kodowanie
 eksploracyjne, 82, 95
komprehencja, 88

komunikat o błędzie, *Patrz:* błąd
 komunikat
konstrukcja try.. except, 40, 41, 44
 błędne zastosowanie, 41
krotka, 74, 84

L

Langr Jeff, 156
liczba losowa, 165
limit czasu HTTP, 62
list comprehension, *Patrz:* lista
 składanie
lista, 74
 indeks, 51, 52
 kontrolna, 167
 składanie, 51
 testowanie, 120
 zagnieżdżona, 72

M

makieta, 134, 135
 time.sleep, 135
Martelli Alex, 41
MazeRun, 28
 grafika, 34, 71, 134
 labirynt, 45, 46, 48
 mechanizm sterowania, 59
Mercurial, 188, 194
metaklasa, 90
metoda
 naukowa, 60, 61, 68
 os.remove, 136
 przetwarzania ciągów znaków,
 73
 Surface.blit, 72
MkDocs, 261
mock, *Patrz:* makieta
model szwajcarskiego sera, 168
moduł
 __builtins__, 88
 csv, 211
 datetime, 236
 difflib, 137
 doctest, 263
 filecmp, 137
 fixtures, 146

moduł

ipdb, *Patrz:* ipdb
 pprint, 74
 pycopg2, 240
 pygame.image, 88
 random, 165
 string, 135
 tempfile, 136
 testowany, 142
 testowy, 142, 147
 tworzenie, 213
 typing, 244
 unittest.mock, 134, 135

N

namespace, *Patrz:* przestrzeń

nazw

narzędzie

fabric, 196
 LaTeX, 253
 make, 194
 mypy, 244, 245
 pydoc, 244
 pylint, *Patrz:* pylint
 pyscaffold, *Patrz:* pyscaffold
 Sphinx, *Patrz:* Sphinx
 virtualenv, *Patrz:* virtualenv

norma

CMMI, 168
 ISO9001, 168
 ITIL, 168

notatnik Jupyter, 262

O

obiekt, 84

opis, 91
 samodokumentujący się, 90
 TemporaryFile, 137
 tożsamość, 93, 94
 typ, 92, *Patrz:* typ, typowanie

operator

+=, 49
 ==, 49, 93
 is, 93, 94

oprogramowanie

dług techniczny, 68
 entropia, 204, 205, 226, *Patrz*
też: kod niezorganizowany
 koszty, 25
 sterowane testami,
Patrz: TDD
 struktury projektu,
Patrz: projekt struktura

Ottinger Tim, 156

P

pakiet

faker, 138, 139
 importowanie, 146, 147
 pytest-cov, 151
 testowy, 147
 virtualenv, 198
 virtualenvwrapper, 198

pandoc, 263

parsowanie, 211

pdb, 99, 109

pętla

sprzężenia zwrotnego, 227, 228
 zdarzeń, 59, 62, 64, 105, 229
 kod, 59
 parametr, 60

plik, 83

.coveragerc, 196
 .gitattributes, 196
 .gitconfig, 185
 .gitignore, 185, 196
 .pdbrc, 105
 .rst, 253
 .travis.yml, 196
 __init__.py, 116, 193
 AUTHORS.rst, 195
 CONTRIBUTING.md, 196
 Dockerfile, 196
 fabfile.py, 196
 fizyczny, 137
 iftex.sty, 253
 konfiguracyjny, 105
 LICENSE.rst, 195
 Makefile, 196
 manage.py, 196

MANIFEST.in, 195

nazwa, 209

odczyt, 135

parsowanie, 211

requirements.txt, 196

setup.py, 195

tekstowy, 204

tox.ini, 196

tymczasowy, 136

versioneer.py, 196

wejściowy, 209

wyjściowy, 209

wynikowy, 137, 138

testowanie, 135

usuwanie, 136

POC, 22, 167

polecenie

!ls, 83

%debug, 84

%env, 84

%hist, 82

%paste, 84

%reset, 84

%run, 84

?naz*, 84

?nazwa, 84

callable, 94

cd, 83, 84

class, 88

commit, 175

debugera, 102, 103, 104

def, 88

del, 88

dir, 85, 86, 87, 91, 93, 94

for, 88

getattr, 94

git add, 182, 184, 187

git bisect, 188

git blame, 188

git branch, 183, 187

git checkout, 178, 179, 187

git cherry pick, 188

git clone, 181, 182, 187

git commit, 177, 181, 182, 184,

187, 210

git config, 187

git diff, 176, 185, 187

- git init, 181, 187
 - git log, 178, 179, 187
 - git merge, 188
 - git mv, 177
 - git pull, 181, 182, 184, 187
 - git push, 182, 184, 187
 - git rebase, 188
 - git rm, 177, 187
 - git status, 175, 184, 185, 187
 - globals, 94
 - hasattr, 94
 - help, 91, 93, 94
 - import, 88, 209, 213
 - import xml, 91
 - ipdb.pm, 101
 - ipdb.set_trace, 100, 101
 - isinstance, 93, 94
 - issubclass, 93, 94
 - list, 94
 - locals, 94
 - ls, 83, 84
 - make, 253
 - mv, 177
 - pdb.set_trace, 103
 - pip install faker, 138
 - pwd, 83, 84
 - py.test, 117, 148
 - rm, 177
 - sys.exc_info, 100
 - Tab, 84
 - tokenizacja, 35
 - type, 92, 93, 94
 - Uniksa, 83
 - with, 88, 101, 211
 - pomysł potwierdzenie, *Patrz:* POC
 - potwierdzenie pomysłu, *Patrz:* POC
 - powłoka
 - IPython, *Patrz:* IPython
 - powłoka
 - Pythona, *Patrz:* Python
 - powłoka
 - program, *Patrz też:* kod architektura, 227, 229
 - wzorzec, *Patrz:* wzorzec architektoniczny
 - zagnieżdżanie, 227
 - implementacja, 231
 - interfejs, *Patrz:* interfejs programu
 - komponent, 229, 230
 - kontekst, 205
 - podział na moduły, 213
 - struktura, 211, 227
 - uruchamianie
 - krokowe, 102, 105, 106
 - w trybie debugowania, 104
 - wydajność, *Patrz:* wydajność
 - wznawianie działania, 103
 - programowanie
 - defensywne, 238, 246
 - dekompozycja zadań, *Patrz:* dekompozycja zadań programowania
 - DRY, 207
 - inżynieria, *Patrz:* inżynieria programowania
 - planowanie, 233
 - rzemiosło, *Patrz:* Software Craftsmanship
 - w parach, 65
 - projekt
 - Cython, *Patrz:* Cython
 - numer wersji, 196
 - struktura, 191, 206, 208, 233
 - plików i katalogów, 192, 193
 - środowisko, 197
 - proof of concept, *Patrz:* POC
 - prototyp, 167, 168
 - przechwytywanie wyjątku, 40, 41, 42, 44
 - przestrzeń nazw, 81, 84, 85, 90, 94
 - atrybut, *Patrz:* atrybut
 - modyfikowanie, 88
 - zagnieżdżanie, 86
 - zasięg, 89
 - pubd, 110
 - pułapka, 98, 103, 104
 - PyCharm, 109, 244
 - pydoc, 262
 - pygments, 262
 - pylint, 215, 245
 - komunikat ostrzegawczy, 215
 - ocena punktowa kodu, 215
 - pyscaffold, 185, 191, 192, 198
 - instalacja, 192
 - numer wersji projektu, 196
 - ograniczenia, 192
 - Python
 - instalacja, 29
 - powłoka, 63
 - wersja, 62
- ## R
- Read the Docs, 262
 - refaktoryzacja, 211, 217
 - wyodrębnianie funkcji, 212
 - repozytorium, 174, 194
 - historia, 178
 - kopia, 187
 - odrzucając zmiany, 177
 - plik, 187
 - dodawanie, 175
 - ignorowanie, 184
 - przenoszenie, 177
 - śledzenie zmian, 176, 177
 - usuwanie, 177
 - tworzenie, 174, 187
 - zdalne, 187
 - rzemiosło programowania, *Patrz:* Software Craftsmanship
- ## S
- Scrum, 25, 220
 - seed, *Patrz:* ziarno
 - Selenium, 157
 - serwer
 - produkcyjny, 62
 - testowy, 62
 - skrypt testowy, 63, 64, 68
 - słownik, 74, 94, 100
 - klucz, 84
 - testowanie, 120
 - Software Craftsmanship, 25, 26
 - software engineering, *Patrz:* inżynieria programowania
 - Sourceforge, 188
 - Sphinx, 194, 250
 - dyrektywa, 254
 - autofunction, 255, 256
 - automodule, 256

Sphinx

- doctest, 256
- ifconfig, 258, 259
- toctree, 254
- todo, 257
- todolist, 258
- konfigurowanie, 250
- szablon, 259
- wartości domyślne, 251

Spolsky Joel, 41

SQLAlchemy, 241

stała, 209

standard PEP8, 216

strategia

- EAFP, *Patrz:* EAFP

- wyszukiwania binarnego, 64

stress test, *Patrz:* test warunków

- skrajnych

SVN, 188

symbol ucieczki, 102

system

- BitBucket, 188

- CVS, 188

- git, 174, 182, 187

- fork, 182

- gałąź, 182, 183, 187

- historia, 178

- kod skrótu, 179

- konfigurowanie, 184, 185

- master, 183

- odrzucać zmiany, 177

- repozytorium, *Patrz:*

- repozytorium

- ustawienia globalne, 185

- żądanie pobrania, 182

GitHub, 179, 180, 181

- projekt, 180, 181, 182

- scalanie zmian, 182

kontrola wersji, 173, 174, 188

- git, *Patrz:* system git

- nierozproszony, 188

Mercurial, 188

Sourceforge, 188

SVN, 188

Ś

środowisko PyCharm, 109, 244

T

tablica NumPy, 239

TDD, 162

test

- akceptacyjny, 156, 157

- automatyczny, 118, 156, 157,

- 158, 163, 167, 216

- ograniczenia, 164, 165, 166

- zalety, 163, 164

- dane, 133

- dokumentacji, 256

FIRST, 156

- integracyjny, 156, 157

- jednostkowy, 156, 157, 257

- niepomyślny, 118, 149

- analiza, 150

- numer, 132

- obciążenia, 156

- parametryzacja, 132, 133, 134

- pisanie, 160, 161

- według defektów, 162

- według specyfikacji, 161

- pomyślny, 118, 149

- projektowanie, 138

- regresji, 156, 157

- sprzątanie, 136

- tworzenie, 116

- uruchamianie, 117, 146, 148,

- 149, 150

- ponowne, 150

- warunków skrajnych, 156

- wydajności, 156, 158

- wykrywanie automatyczne,

- 148

- zdrowego rozsądku, 131

- zestaw, 141, 142, 148, 153

test coverage, *Patrz:* testowanie

- pokrycie testami

test suite, *Patrz:* test zestaw

Test-Driven Development,

- Patrz:* TDD

testowanie, 147, 160

- automatyczne, 115, 118, 125,

- 156, 163, 167

- ograniczenia, 164, 165, 166

- zalety, 163, 164

- danych, *Patrz:* dane testowe

plików wynikowych, 135

pokrycie testami, 141, 151,

- 152, 153

występowania wyjątków, 121,

- 122

timeout, *Patrz:* limit czasu HTTP

Torvalds Linus, 174

Tox, 157, 200

traceback, *Patrz:* błąd ślad

typ, 81

- adnotacja, 242, 243

- kontrola, 237, 238, 239, 240,

- 241, 242, 243, 244

- predefiniowany, 244

- złożony, 244

- znaczenie semantyczne, 236

type hints, *Patrz:* typowanie

- wskazówki

typowanie

- dynamiczne, 235

- wady, 236, 237

- statyczne, 237, 241, 243

- wskazówki, 243, 244

U

Unicode, 38

V

virtualenv, 197, 198, 199, 200

W

wcięcie, 36, 54

wdb, 111

wielowątkowość, 229

współbieżność, 166, 222, 225, 227

wydajność, 158, 159

wyjątek, 33, *Patrz też:* błąd

- TypeError, 42

- AssertionError, 118, 120, 238

- AttributeError, 42

- IndentationError, 35, 36, 37

- IndexError, 42

- informacje, 100

- IOError, 40, 41

- KeyError, 38, 40, 100

- wyjątek
 - NameError, 56, 88
 - przechwytywanie, 40, 41, 42, 44
 - testowanie, *Patrz*: testowanie występowania wyjątków
 - ValueError, 42
 - wywoływany podczas działania kodu, 35
 - ZeroDivisionError, 76
 - zgłaszany przed uruchomieniem kodu, 35, *Patrz też*: błąd SyntaxError
 - wymagania
 - analiza, 219, 220
 - funkcjonalne, 223, 225
 - niefunkcjonalne, 224, 225
 - wyrazenie, 102
 - if, 213
 - logiczne, 54
 - warunkowe, 78
 - wywołanie zwrotne, 60, 105, 231
 - wzorzec architektoniczny
 - antywzorzec, 227
 - blob, 227
 - mediator, 227, 228, 229
 - model warstw, 227
 - pętla sprzężenia zwrotnego, 227, 228
 - pieluchy, 41
 - potok, 227
 - spaghetti, 227
 - zagnieżdżanie, 227
- X**
- XP, 25
- Z**
- zasada DRY, 207
 - zbiór, 74
 - zdarzenie
 - klawiatury, 59
 - myszy, 59
 - niestandardowe, 231
 - pętla, *Patrz*: pętla zdarzeń
 - Zeller Andreas, 56
 - ziarno, 61, 165
 - zmienna, 73
 - definicja, 244
 - ls, 83
 - nadpisanie, 48
 - nazwa, 209, 210
 - przypisanie wartości, 48
 - przypadkowe, 49
 - PYTHONPATH, 116, 201, 255
 - środowiskowa, 84
 - falszywa, 135
 - PYTHONPATH, *Patrz*: zmienna PYTHONPATH
 - zasięg, 89
 - znak
 - #, 38, 40
 - dwukropka, 36
 - końca wiersza, 49
 - lewego ukośnika, 42
 - podkreślenia, 86
 - tabulacji, 36
 - wykrzyknika, 83, 102

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>

Ty też możesz programować w Pythonie!

Python istnieje już ponad 25 lat. Nie jest trudnym językiem i oferuje ogromne możliwości. Tworzenie dobrych programów w Pythonie wymaga jednak od programistów dużych umiejętności. Cykl rozwoju oprogramowania jest pełen pułapek nieznanym początkującym koderom. Mimo to w podręcznikach Pythona niewiele uwagi poświęca się debugowaniu i testowaniu, a przecież etapy te mają kluczowe znaczenie dla jakości kodu i funkcjonalności tworzonego oprogramowania.

Ta książka uczyni Cię lepszym programistą! Dzięki przedstawionym tu podstawowym praktykom stosowanym przez najbardziej profesjonalnych programistów Pythona będziesz tworzył doskonalszy kod. Zoptymalizujesz procesy debugowania programów, pisania automatycznych testów i utrzymywania oprogramowania bez nadmiernego wysiłku. Przedstawione tu techniki będą szczególnie przydatne dla programistów zajmujących się analizą danych, tworzeniem stron internetowych oraz rozwijaniem oprogramowania naukowego.

Najważniejsze zagadnienia przedstawione w książce:

- błędy semantyczne i wyjątki
- sposoby eliminacji błędów i narzędzia do debugowania
- zasady i techniki testowania aplikacji
- mocne i słabe strony testów automatycznych
- mechanizm kontroli wersji

Dr Kristian Rother — zajmuje się programowaniem od wczesnego dzieciństwa. Jest również specjalistą w dziedzinie bioinformatyki: prowadził badania struktur 3D białek i RNA na Uniwersytecie Humboldta w Berlinie. Przez wiele lat doskonalił swoje umiejętności nauczania, a obecnie pracuje jako profesjonalny trener. Prowadzi szkolenia z programowania w Pythonie, uczy biochemii, statystyki, testowania aplikacji internetowych, wyszukiwarek, wygłasza prezentacje.



KOD KORZYŚCI

ISBN 978-83-283-3802-9
9 788328 338029

cena: 49,00 zł

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Informatyka w najlepszym wydaniu

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowości>

Apress