# 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



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.

# Spis treści

ROZDZIAŁ 2

# Wyjątki w Pythonie

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

— 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.

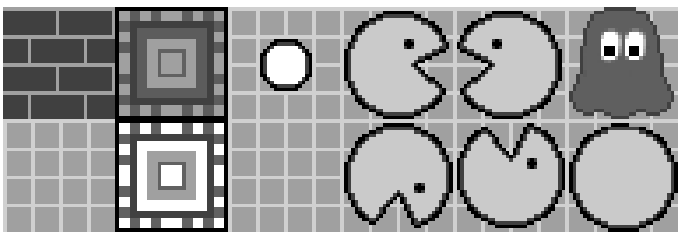
## Wyjątki są defektami, o których istnieniu wiemy

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.


***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`:

```
imprt 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
    imprt 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 podpowiedzi, że brakuje zamykającego nawiasu kwadratowego. Doszedł do końca pliku i zakończył działanie. Gdybyśmy jednak dodali nową linijkę 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
```

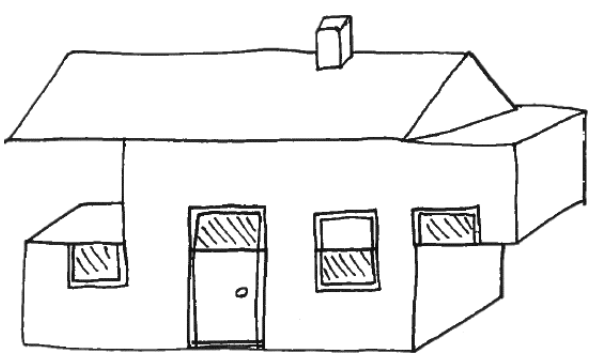
Python nie wie, dlaczego w powyższym kodzie znajduje się wcięcie w drugim wierszu. Zwróćmy uwagę, że wyjątek `IndentationError` powstaje tylko wtedy, gdy pierwszy element listy jest podawany w wierszu, w którym znajduje się instrukcja przypisania. W przeciwnym razie ponownie uzyskamy błąd `SyntaxError`. 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 przeznaczonego 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, utwórzmy 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, przyjrzymy 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.

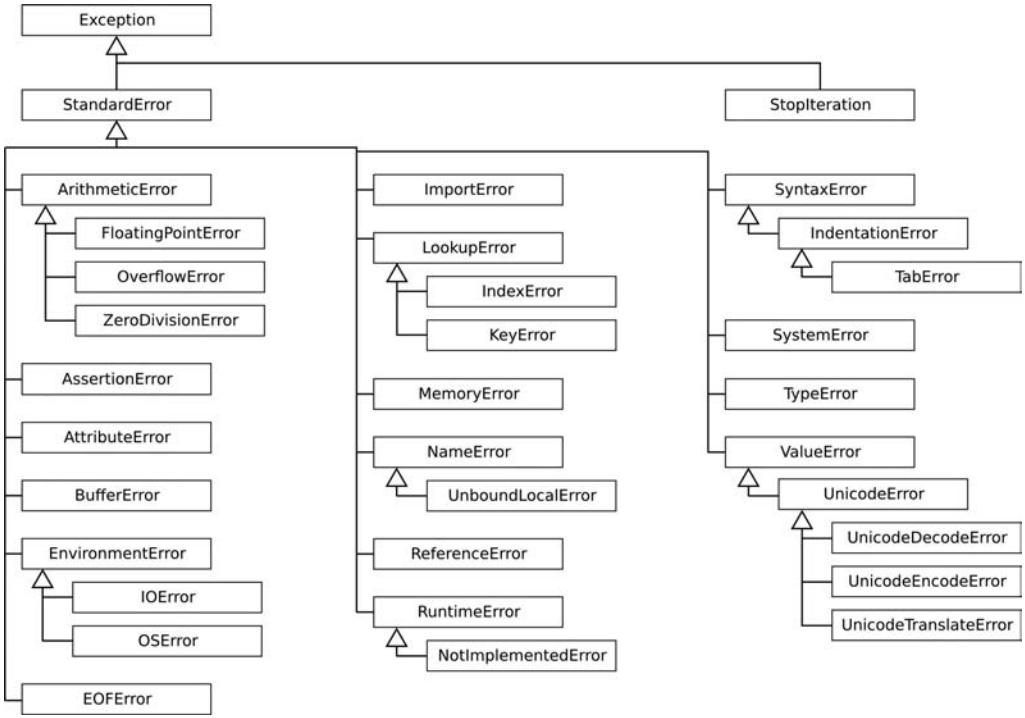
Od wydania Pythona 3 i wprowadzenia *Unicode* liczba możliwości popełnienia błędów w pisowni znaków zwiększyła się o kilka rzędów wielkości. Znajomość możliwych typów wyjątków oraz ich znaczenia to solidna wiedza podstawowa dla doświadczonych programistów Pythona. W naszym przypadku wystąpienie wyjątku `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 przejrzenia 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 zaczynać 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('tiless.xpm', 'rb')
```

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

```
FileNotFoundError: [Errno 2] No such file or directory: 'tiless.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))
```

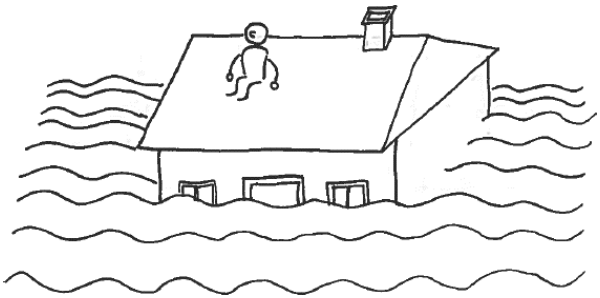
Instrukcja `except` umożliwia zdefiniowanie właściwej reakcji na specyficzne typy wyjątków. Strategię tę nazwano **EAFP** (od ang. *Easier to Ask Forgiveness than Permission* — dosł. łatwiej poprosić o wybaczenie niż o zgodę). *Prośba o przebaczenie* oznacza zareagowanie na wyjątek, *prośba o zgodę* 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 naprawdę tego chcieliśmy?*

## 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 wyciszaniem wyjątków za pomocą konstrukcji `try.. except`? Przechwytywanie wyjątków to świetna strategia radzenia sobie z sytuacjami *wyjątkowymi*, 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 działać 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 dwukropek, 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, zapomniałem 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 pomocna 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 patrzenie 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.

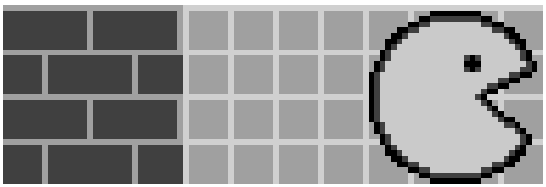
# Poprawny kod

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

## E



## F



## G



## H

## I



## J



## K



## L



## M

## N



## O



## P

## R



## S

## Ś



## T



## U



## V



## W

## X



## Z

# 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 nieznanych 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.




ISBN 978-83-283-3802-9