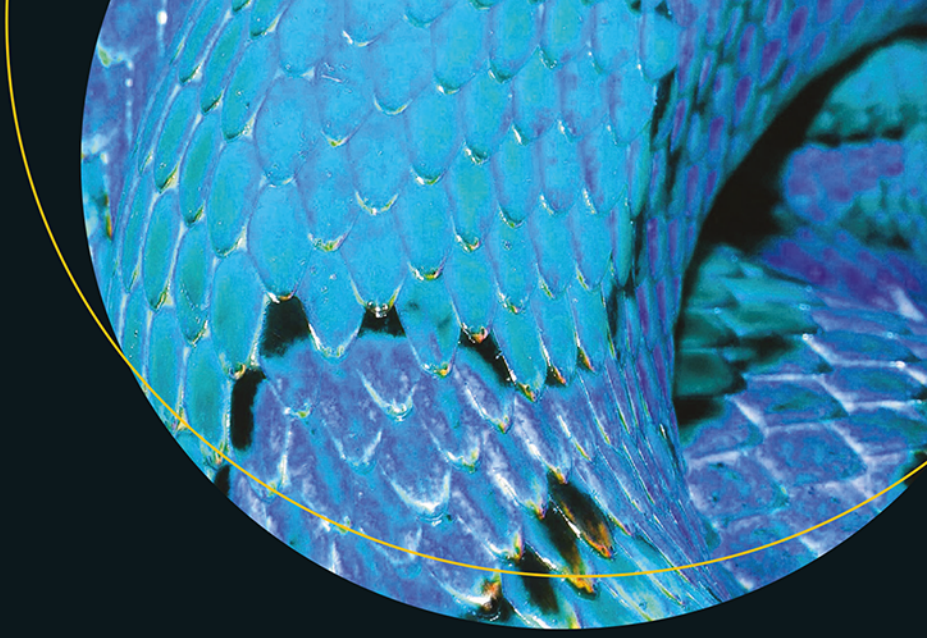

# Czysty kod w Pythonie

Sunil Kapil

Helion

Apress

Tytuł oryginału: Clean Python: Elegant Coding in Python

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-6462-2

First published in English under the title Clean Python; Elegant Coding in Python by Sunil Kapil, edition: 1


Helion SA
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/czykop*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*ftp://ftp.helion.pl/przyklady/czykop.zip*

Printed in Poland.

# Spis treści

ROZDZIAŁ 1

# Kodowanie pythoniczne

Tym, co odróżnia Pythona od innych języków programowania, jest prostota, ale o dużej głębi. Z racji prostoty tego języka bardzo ważne jest, aby kod pisać uważnie, szczególnie w dużych projektach, ponieważ można go łatwo skomplikować i nadmiernie rozbudować. Python ma swoją filozofię zen, która przedkłada prostotę nad złożoność[1].

W tym rozdziale poznasz kilka popularnych praktyk, dzięki którym Twój kod będzie prostszy i bardziej czytelny. Opisanych jest tutaj kilka znanych i mniej znanych technik. Pamiętaj o nich podczas pracy nad obecnymi i przyszłymi projektami, a Twój kod będzie lepszy.

---

■ **Uwaga** Jeżeli będziesz postępował zgodnie z filozofią zen Pythona, Twój kod będzie „pythoniczny". W oficjalnej dokumentacji tego języka opisanych jest mnóstwo dobrych praktyk, dzięki którym kod jest bardziej przejrzysty i czytelny. Lektura dokumentacji PEP8 z pewnością pomoże Ci zrozumieć, dlaczego stosowanie niektórych praktyk jest szczególnie zalecane.

---

## Tworzenie pythonicznego kodu

Oficjalna dokumentacja PEP8, dostępna na stronie *https://www.python.org/dev/peps/pep-0008*, opisuje wiele dobrych praktyk pisania kodu w języku Python, które od czasu ich powstania uległy dużym zmianom. W tym rozdziale skupimy się na kilku z nich oraz na korzyściach, jakie ma z nich programista.

---

[1] *https://www.python.org/dev/peps/pep-0020*

# Nazewnictwo

Jestem programistą kodującym w różnych językach, m.in. Java, NodeJS, Perl i Golang. W każdym z nich obowiązują konwencje nazewnictwa zmiennych, funkcji, klas itp. W Pythonie również jest zalecane stosowanie pewnych konwencji. W tej części rozdziału omówionych jest kilka z nich, których powinieneś przestrzegać, pisząc swój kod.

## Zmienne i funkcje

W nazwach zmiennych i funkcji stosuj małe litery, a słowa oddzielaj znakiem podkreślenia. Dzięki temu będą bardziej czytelne, jak w listingu 1.1.

***Listing 1.1.*** *Nazwy zmiennych*

```
names = "Python"                    # Nazwa zmiennej
job_title = "Programista"           # Nazwa zmiennej ze znakiem podkreślenia
populated_countries_list = []       # Nazwa zmiennej
                                    # ze znakiem podkreślenia
```

Stosuj również niekonfliktowe nazwy, umieszczając przed nimi jeden lub dwa znaki podkreślenia, jak w listingu 1.2.

***Listing 1.2.*** *Niekonfliktowe nazwy*

```
_books = {}      # Definicja prywatnej zmiennej
__dict = []      # Nazwa niekolidująca z innymi nazwami we wbudowanych bibliotekach
```

Pojedynczy znak podkreślenia stosuj jako prefiks w nazwach wewnętrznych zmiennych, do których nie mogą mieć dostępu inne klasy. To jest tylko konwencja, ponieważ umieszczenie pojedynczego znaku podkreślenia nie definiuje zmiennej jako prywatnej.

W nazwach funkcji też obowiązują konwencje pokazane w listingu 1.3.

***Listing 1.3.*** *Nazwy zwykłych funkcji*

```
def get_data():
    ...
def calculate_tax_data():
    ...
```

W przypadku metod obowiązują takie same zasady pozwalające uniknąć konfliktu nazw z wbudowanymi funkcjami (patrz listing 1.4).

***Listing 1.4.*** *Niekonfliktowe nazwy prywatnych metod*

```
# Prywatna metoda z pojedynczym znakiem podkreślenia
def _get_data():
    ...
# Podwójny znak podkreślenia zapobiegający konfliktowi z nazwami wbudowanych funkcji
def __path():
    ...
```

Oprócz przestrzegania powyższych zasad ważne jest nadawanie funkcjom i zmiennym zrozumiałych nazw.

Przeanalizujmy przykład funkcji zwracającej obiekt użytkownika o danym identyfikatorze (patrz listing 1.5).

***Listing 1.5.*** *Nazwy funkcji*

```
# Zła nazwa
def get_user_info(id):
    db = get_db_connection()
    user = execute_query_for_user(id)
    return user

# Dobra nazwa
def get_user_by(user_id):
    db = get_db_connection()
    user = execute_user_query(user_id)
    return user
```

Przeznaczenie drugiej funkcji, `get_user_by()`, jest oczywiste, ponieważ jej nazwa odnosi się do argumentu. Natomiast nazwa pierwszej funkcji, `get_user_info()`, jest niezrozumiała, ponieważ argument może oznaczać cokolwiek, na przykład indeks użytkownika, identyfikator płatności lub coś innego. Tego rodzaju kod będzie niezrozumiały dla innych programistów. Aby go poprawić, zmieniłem nazwę funkcji i jej argumentu, dzięki czemu kod stał się bardziej czytelny. Teraz od razu wiadomo, jakie jest przeznaczenie drugiej funkcji i jaki zwraca ona wynik

Twoim obowiązkiem jest staranne dobieranie nazw zmiennych i funkcji, aby kod był czytelny dla innych programistów.

## Klasy

Tak jak w większości języków, nazwy klas powinny składać się z wielkich i małych liter. W listingu 1.6 przedstawiono prosty przykład.

***Listing 1.6.*** *Nazwy klas*

```
class UserInformation:
    def get_user(id):
        db = get_db_connection()
        user = execute_query_for_user(id)
        return user
```

### Stałe

Nazwy stałych powinny składać się wyłącznie z wielkich liter, jak w przykładowym listingu 1.7.

***Listing 1.7.*** *Nazwy stałych*

```
TOTAL = 56
TIMOUT = 6
MAX_OVERFLOW = 7
```

### Nazwy argumentów funkcji i metod

Zasady nazewnictwa argumentów funkcji i metod są takie same jak w przypadku zmiennych i metod. Metoda różni się od funkcji tym, że jej pierwszym argumentem jest `self` (patrz listing 1.8).

***Listing 1.8.*** *Nazwy argumentów funkcji i metod*

```
def calculate_tax(amount, yearly_tax):
    ...
class Player:
    def get_total_score(self, player_name):
        ...
```

## Wyrażenia i instrukcje

Zapewne w celu zmniejszenia liczby wierszy i zaimponowania kolegom próbowałeś pisać kod na różne zmyślne sposoby. Jednak na takim podejściu traci czytelność i prostota kodu. Przeanalizujmy kod z listingu 1.9 sortującego zagnieżdżony słownik.

***Listing 1.9.*** *Sortowanie zagnieżdżonego słownika*

```
users = [{"first_name":"Helena", "age":39},
         {"first_name":"Bartek", "age":10},
         {"first_name":"Ania", "age":9}
        ]
users = sorted(users, key=lambda user: user["first_name"].lower())
```

Co jest złego w tym kodzie? Za pomocą jednowierszowej funkcji lambda słownik jest sortowany według klucza `first_name`. Jest to bardziej pomysłowy sposób niż stosowanie pętli. Jednak trudno jest, szczególnie innemu programiście, ocenić na pierwszy rzut oka, co się w kodzie dzieje. Powodem jest funkcja lambda, która nie jest prostym pojęciem, a do tego ma skomplikowaną składnię. Oczywiście dzięki niej można pomysłowo posortować słownik, a kod składa się z mniejszej liczby wierszy, ale to nie oznacza, że jest poprawny i czytelny. Nie będzie działał poprawnie, jeżeli słownik będzie zawierał błędne wartości lub zabraknie w nim kluczy. Aby kod działał poprawnie i był bardziej czytelny, zastosujmy w nim funkcję, która będzie sprawdzała, czy wartości są poprawne. Funkcja ta jest bardzo prosta (patrz listing 1.10).

***Listing 1.10.*** *Sortowanie słownika za pomocą funkcji*

```
users = [{"first_name":"Helena", "age":39},
         {"first_name":"Bartek", "age":10},
         {"first_name":"Ania", "age":9}
        ]
def get_user_name(users):
    """Odczytanie imienia i zamiana liter na małe."""
    return users["first_name"].lower()
def get_sorted_dictionary(users):
    """Sortowanie zagnieżdżonego słownika."""
    if not isinstance(users, dict):
        raise ValueError("Błędna wartość w słowniku")
    if not len(users):
        raise ValueError("Słownik jest pusty")
    users_by_name = sorted(users, key=get_user_name)
    return users_by_name
```

Powyższy kod sprawdza poprawność wartości i jest o wiele bardziej czytelny niż w poprzedniej wersji z jednowierszowym wyrażeniem, które wprawdzie zmniejszało liczbę wierszy, ale znacznie komplikowało kod. Nie oznacza to oczywiście, że wyrażenia jednowierszowe są złe. Uważam jednak, że należy unikać ich stosowania, jeżeli pogarszają czytelność kodu.

Podczas pisania kodu decyzje takie jak powyższa będziesz podejmował nieustannie. Czasami dzięki jednowierszowym wyrażeniom kod jest bardziej czytelny, czasami mniej.

Rozważmy przykład kodu odczytującego zawartość pliku CSV i zliczającego jego wiersze. Listing 1.11 pokazuje, dlaczego ważna jest czytelność kodu i stosowanie w nim zrozumiałych nazw. Dzięki wyodrębnieniu funkcji pomocniczej kod jest bardziej czytelny, a w środowisku produkcyjnym łatwiejszy w diagnozowaniu problemów.

***Listing 1.11.*** *Odczytywanie pliku CSV*

```
import csv
with open("pracownicy.csv", mode="r") as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Nazwy kolumn: {", ".join(row)}')
            line_count += 1
            print(f'\t{row["name"]} wynagrodzenie: {row["salary"]} '
                  f'data urodzenia {row["birthday"]}.')
        line_count += 1
    print(f'Przetworzono {line_count} wierszy.')
```

Wewnątrz instrukcji `with` wykonywanych jest kilka operacji. Aby kod był bardziej czytelny i mniej podatny na błędy, można fragment przetwarzający zawartość pliku wyodrębnić do postaci osobnej funkcji. Trudno diagnozować kod, w którym w kilku wierszach jest wykonywanych wiele operacji, dlatego warto tworzyć funkcje o jasno określonym przeznaczeniu. Listing 1.12 przedstawia kod podzielony na części.

***Listing 1.12.*** *Bardziej czytelny kod przetwarzający plik CSV*

```
import csv
def process_salary(csv_reader):
    """Przetworzenie pliku CSV."""
    global line_count
    for row in csv_reader:
        if line_count == 0:
            print(f'Nazwy kolumn: {", ".join(row)} '
                  f'data urodzenia {row["birthday"]}.')
            line_count += 1
        print(f'\t{row["name"]} wynagrodzenie: {row["salary"]}')
        line_count += 1
    print(f'Przetworzono {line_count} wierszy.')

with open("pracownicy.csv", mode="r") as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    process_salary(csv_reader)
```

Powyższy kod zamiast wykonywać wszystkie operacje wewnątrz instrukcji `with` wykorzystuje funkcję pomocniczą. Teraz widać wyraźnie, co robi funkcja `process_salary()`. Gdyby trzeba było obsłużyć określony wyjątek lub przetworzyć inne dane zawarte w pliku CSV, można powyższą funkcję podzielić na kolejne części zgodnie z zasadą pojedynczej odpowiedzialności.

# Pythoniczny styl kodowania

Dokumentacja PEP8 zawiera zalecenia, dzięki którym kod jest bardziej przejrzysty i czytelny. Przeanalizujmy kilka z nich.

## Stosuj metodę join() zamiast operatora łączenia ciągów znaków

Jeżeli ważna jest wydajność kodu, stosuj metodę `join()` zamiast operatora łączenia ciągów, np. `a += b` lub `a = a + b` (patrz listing 1.13). Metoda `join()`gwarantuje szybsze łączenia ciągów w różnych implementacjach Pythona, ponieważ przydziela pamięć na połączone ciągi tylko raz. Natomiast w przypadku operatora pamięć jest przydzielana dla każdego ciągu osobno, ponieważ w Pythonie ciągi są niemutowalne.

***Listing 1.13.*** *Użycie metody join()*

```
first_name = "Jan"
last_name = "Nowak"
# Niezalecany sposób łączenia ciągów.
full_name = first_name + "  " + last_name
# Bardziej wydajny i czytelny sposób.
" ".join([first_name, last_name])
```

## Do sprawdzania wartości None stosuj instrukcje is oraz is not

Do sprawdzania wartości `None` stosuj instrukcje `is` oraz `is not`. Pamiętaj o tym, pisząc na przykład następujący kod:

```
if val:     # Instrukcja nie będzie działać poprawnie, jeżeli val będzie miała wartość None
```

Zmienna `val` może być typu kontenerowego, na przykład `dict` lub `set`. Przeanalizujmy dokładniej przypadki, które mogą Cię zaskoczyć.

Jeżeli w powyższym przykładzie zmienna `val` będzie zawierała pusty słownik, zostanie potraktowana tak, jakby zawierała wartość `false`, co może nie być pożądanym efektem. Dlatego zachowaj ostrożność, pisząc kod w ten sposób.

**Nie rób tak:**

```
val = {}
if val:                 # Wyrażenie przyjmie wartość false
```

Zawsze warunki definiuj jak najściślej, dzięki czemu kod będzie mniej podatny na błędy.

**Rób tak:**

```
if val is not None:  # Wyrażenie przyjmie wartość false tylko wtedy, gdy zmienna będzie miała
                     # wartość None
```

## Stosuj warunek is not zamiast not … is

Nie ma różnicy pomiędzy warunkami `is not` a `not ... is`. Jednak pierwszy zapis jest bardziej czytelny.

**Nie rób tak:**

```
if not val is None:
```

**Tylko tak:**

```
if val is not None:
```

## W instrukcjach przypisania stosuj zwykłą funkcję zamiast funkcji lambda

W instrukcji przypisania stosuj zwykłą funkcję zamiast funkcji lambda. Za pomocą słowa kluczowego `lambda` można kodować operacje w jednym wierszu. Czasami jednak lepszym rozwiązaniem jest zdefiniowanie funkcji za pomocą instrukcji `def`.

**Nie rób tak:**

```
square = lambda x: x * x
```

**Tylko tak:**

```
def square(val):
    return val * val
```

Wynik funkcji `square()` łatwiej jest przekształcić w ciąg znaków, jak również łatwiej jest diagnozować tę funkcję niż lambda. Z tego powodu wyrażenia tego rodzaju przestają być przydatne. Funkcję lambda stosuj w większych wyrażeniach, o ile nie wpłynie to na czytelność kodu.

## Konsekwentnie stosuj instrukcję return

Jeżeli funkcja zwraca wynik, sprawdzaj, czy robi to we wszystkich swoich wątkach. Dobrą praktyką jest umieszczanie instrukcji `return` w każdym miejscu, w którym funkcja kończy działanie. Funkcje, które jedynie wykonują określone operacje, domyślnie zwracają wynik `None`.

**Nie rób tak:**

```
def calculate_interest(principle, time, rate):
    if principle > 0:
        return (principle * time * rate) / 100
def calculate_interest(principle, time, rate):
    if principle < 0:
        return
    return (principle * time * rate) / 100
```

**Tylko tak:**

```
def calculate_interest(principle, time, rate):
    if principle > 0:
        return (principle * time * rate) / 100
    else:
        return None
def calculate_interest(principle, time, rate):
    if principle < 0:
        return None
    return (principle * time * rate) / 100
```

## Stosuj metody startswith() i endswith()

W instrukcjach sprawdzających prefiksy i sufiksy ciągów stosuj metody `startswith()` i `endswith()` zamiast wyrażeń wyodrębniających fragmenty ciągu. Instrukcja wyodrębniania fragmentu ciągu jest bardzo przydatna, szczególnie w przypadku przetwarzania dłuższych ciągów lub wykonywania na nich dodatkowych operacji. Jeżeli trzeba po prostu sprawdzić prefiks lub sufiks, lepiej stosować metody `startswith()` i `endswith()`, ponieważ kod jest wtedy bardziej zrozumiały.

**Nie rób tak:**

```
data = "Witaj, co słychać?"
if data[:5] == "Witaj":
```

**Tylko tak:**

```
data = "Witaj, co słychać?"
if data.startswith("Witaj"):
```

## Sprawdzaj typy za pomocą metody isinstance(), a nie type()

Do sprawdzania typu danych używaj metody `isinstance()`, a nie `type()`, ponieważ tę pierwszą można stosować również w przypadku podklas. Przeanalizujmy przypadek struktury będącej podklasą `dict`, na przykład `orderdict`. Metoda `type()` nie będzie działać poprawnie, natomiast `isinstance()` prawidłowo rozpozna, że struktura jest podklasą `dict`.

**Nie rób tak:**

```
user_ages = {"Leszek": 35, "Joanna": 89, "Igor": 12}
if type(user_ages) == dict:
```

**Tylko tak:**

```
user_ages = {"Leszek": 35, "Joanna": 89, "Igor": 12}
if isinstance(user_ages, dict):
```

## Porównywanie wartości logicznych

Wartości logiczne można porównywać na kilka sposobów.

**Nie rób tak:**

```
is_empty = False
if is_empty == False:
if is_empty is False:
```

**Tylko tak:**

```
is_empty = False
if not is_empty:
```

## Pisz ścisły kod dla menedżera kontekstu

Jeżeli używasz instrukcji `with`, twórz osobne funkcje wykonujące operacje początkowe, główne i końcowe.

**Nie rób tak:**

```
class NewProtocol:
    def __init__(self, host, port, data):
        self.host = host
        self.port = port
        self.data = data
    def __enter__(self):
        self._client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self._client.connect((self.host, self.port))
        self._transfer_data()
    def __exit__(self, exception, value, traceback):
        self._receive_data()
        self._client.close()
    def _transfer_data(self):
        ...
    def _receive_data(self):
        ...
con = NewProtocol(host, port, data)
```

**Tylko tak:**

```
class NewProtocol:
    def __init__(self, host, port):
        self.host = host
        self.port = port
    def __enter__(self):
        self._client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self._client.connect((self.host, self.port))
    def __exit__(self, exception, value, traceback):
        self._client.close()
    def transfer_data(self, payload):
       ...
    def receive_data(self):
       ...
with NewProtocol(host, port) as con:
    transfer_data()
    receive_data()
```

Metody `__enter__()` i `__exit__()` w pierwszej wersji oprócz otwierania i zamykania połączenia wykonują również dodatkowe operacje. Lepszym rozwiązaniem jest tworzenie dla tych operacji osobnych metod.

## Stosuj narzędzia weryfikujące poprawność kodu

Narzędzia weryfikujące poprawność kodu ułatwiają utrzymanie go w spójnym formacie, co jest bardzo ważne, szczególnie w skali całego projektu. Narzędzia te analizują kod pod kątem następujących cech:

- błędów składniowych;
- niewykorzystywanych zmiennych i niewłaściwych argumentów funkcji;
- niezgodności z zaleceniami PEP8.

Dzięki narzędziom weryfikującym poprawność kodowanie jest znacznie wydajniejsze, ponieważ na bieżąco wyszukiwane są różne nieścisłości. Dla języka Python dostępnych jest wiele takich narzędzi. Niektóre z nich obejmują określone obszary, na przykład style komentarzy dokumentacyjnych. Narzędzia takie jak Flake8 lub Pylint weryfikują kod pod kątem wszystkich zaleceń PEP8, inne — tylko wybranych. Na przykład mypy sprawdza typy danych. Możesz używać wszystkich lub tylko wybranych, przeprowadzających standardową weryfikację pod kątem zaleceń PEP8. Najlepsze narzędzia to Flake8 i Pylint. Zawsze upewniaj się, czy narzędzie, którego używasz, weryfikuje kod pod kątem zaleceń PEP8.

Narzędzie weryfikujące powinno sprawdzać następujące aspekty kodu:

- zgodność z zaleceniami PEP8;
- kolejność importowanych modułów;
- konwencje nazw zmiennych, funkcji, klas, modułów, plików itp.;
- zapętlone instrukcje importujące;
- złożoność (liczba wierszy kodu funkcji, pętli i innych struktur);
- poprawność pisowni;
- styl komentarzy dokumentacyjnych.

Narzędzia weryfikujące stosuje się na różne sposoby, m.in.:

- podczas kodowania w środowisku IDE;
- podczas zatwierdzania kodu;
- w systemach ciągłej integracji, np. Jenkins lub CircleCI.

---

**Uwaga** Istnieją różne dobre praktyki, dzięki którym zdecydowanie można podnieść jakość kodu. Jeżeli chcesz je jak najpełniej wykorzystać, zapoznaj się z oficjalną dokumentacją PEP8. Pomocne jest również przeglądanie dobrego kodu umieszczonego w serwisie GitHub.

---

# Komentarze dokumentacyjne

Komentarze doskonale nadają się do tworzenia dokumentacji kodu w języku Python. Komentarz dokumentacyjny umieszcza się zazwyczaj na początku kodu metody, kolumny lub modułu. Jest on specjalnym atrybutem `__doc__` danego obiektu. Zgodnie z PEP8 komentarz dokumentacyjny umieszcza się wewnątrz potrójnych cudzysłowów (patrz listing 1.14). Poznaj kilka dobrych praktyk umieszczania komentarzy w kodzie.

***Listing 1.14.*** *Funkcja z komentarzem dokumentacyjnym*

```
def get_prime_number():
    """Lista liczb pierwszych z zakresu od 1 do 100."""
```

Komentarze dokumentacyjne można wpisywać na różne sposoby (opisane w dalszej części rozdziału), jednak zgodnie z kilkoma wymienionymi niżej zasadami:

- Potrójny cudzysłów stosuje się nawet wtedy, gdy komentarz zajmuje tylko jeden wiersz. Jest to dobra praktyka umożliwiająca rozszerzanie komentarza.
- Przed komentarzem i po nim nie pozostawia się pustych wierszy.
- Na końcu komentarza umieszcza się kropkę.

Podobne zasady obowiązują podczas umieszczania wielowyrazowych komentarzy dokładniej opisujących kod (patrz listing 1.15). Dzięki temu dokumentacja kodu jest umieszczana w nim samym i nie trzeba korzystać z osobnych, długich i nużących opisów.

***Listing 1.15.*** *Wielowierszowy komentarz dokumentacyjny*

```
def call_weather_api(url, location):
    """Informacje o pogodzie w zadanym miejscu.

    Funkcja weather_api() zwraca informacje o pogodzie w zadanym miejscu,
    wykorzystując zadany interfejs API. W argumencie location można umieszczać
    jedynie nazwy miast. Umieszczenie nazwy państwa lub regionu
    spowoduje zgłoszenie wyjątku.
    :param url: Adres interfejsu API z informacjami o pogodzie.
    :type url: str
    :param location: Nazwa miasta.
    :type location: str
    :return: Informacje o pogodzie w zadanym mieście.
    :rtype: str
    """
```

Zwróć uwagę na kilka następujących szczegółów:

- Pierwszy wiersz zawiera krótki opis funkcji lub klasy.
- Na końcu wiersza jest kropka.
- Pomiędzy krótkim a pełnym opisem znajduje się pusty wiersz.

Ten sam kod napisany w języku Python 3 z modułem `typing` wyglądałby jak w listingu 1.16.

***Listing 1.16.*** *Wielowierszowy komentarz dokumentacyjny i moduł typing*

```
def call_weather_api(url: str, location: str) -> str:
    """Informacje o pogodzie w zadanym miejscu.

    Funkcja weather_api() zwraca informacje o pogodzie w zadanym miejscu,
    wykorzystując zadany interfejs API. W argumencie location można umieszczać
    jedynie nazwy miast. Umieszczenie nazwy państwa lub regionu
    spowoduje zgłoszenie wyjątku."""
```

Jeżeli w kodzie stosujesz nazwy typów, nie musisz w komentarzu opisywać argumentów.

Jak wspomniałem wcześniej, na przestrzeni lat wypracowano kilka stylów komentarzy dokumentacyjnych, z których żaden nie jest bardziej zalecany lub lepszy od innych. Ważne jest zatem, aby wybrany styl stosować konsekwentnie, dzięki czemu dokumentacja będzie miała spójny format.

Komentarze dokumentacyjne można wpisywać na kilka sposobów.

- Stosowany przez Google:

```
"""Wysłanie żądania na określony adres URL.
Parameters:
    url (str): Adres URL, na który wysyłane jest żądanie.
Returns:
    dict: Odpowiedź zwrócona przez interfejs API.
"""
```

- Strukturalny, zalecany w oficjalnej dokumentacji:

```
"""Wysłanie żądania na określony adres URL.
:param url: Adres URL, na który wysyłane jest żądanie.
:type url: str
:returns: Odpowiedź zwrócona przez interfejs API.
:rtype: dict
"""
```

- Przyjęty w narzędziach NumPy i SciPy:

```
"""Wysłanie żądania na określony adres URL.
Parameters
----------
url : str
    Adres URL, na który wysyłane jest żądanie.
Returns
-------
dict
    Odpowiedź zwrócona przez interfejs API.
"""
```

- Przyjęty w narzędziu Epytext:

```
"""Wysłanie żądania na określony adres URL.
@type url: str
@param file_loc: Adres URL, na który wysyłane jest żądanie.
@rtype: dict
@returns: Odpowiedź zwrócona przez interfejs API.
"""
```

## Komentarze dokumentacyjne do modułów

Komentarz dokumentacyjny opisujący zastosowanie modułu umieszcza się na początku pliku, przed instrukcjami `import` (patrz listing 1.17). Komentarz powinien zawierać przeznaczenie modułu i zawartych w nim wszystkich klas i metod. Nie powinien skupiać się na określonych klasach i metodach. Niektóre z nich można krótko opisać, jeżeli użytkownik powinien się czegoś o nich dowiedzieć przez użyciem modułu.

***Listing 1.17.*** *Modułowy komentarz dokumentacyjny*

```
"""
Moduł zawiera wszystkie potrzebne wywołania sieciowe. Kontroluje wszystkie
zgłaszane przez wywołania wyjątki i zgłasza własne w sytuacjach nieprzewidzianych.
Kod wykorzystujący ten moduł musi obsługiwać następujące wyjątki:
NetworkError: błąd sieciowy.
NetworkNotFound: nieznany adres sieciowy.
"""
import urllib3
import json
...
```

Podczas wpisywania modułowych komentarzy dokumentacyjnych pamiętaj o następujących zasadach:

- Krótko opisuj przeznaczenie modułu.
- Możesz umieszczać dokładniejsze informacje, na przykład o wyjątkach, jak w listingu 1.17, jeżeli uznasz, że są one ważne dla użytkownika. Nie wchodź jednak zbyt głęboko w szczegóły.
- Komentarz powinien zawierać ogólne informacje o module, bez szczegółowego opisywania każdej zawartej w nim funkcji lub operacji wykonywanych przez klasy.

## Komentarze dokumentacyjne do klas

Komentarz dokumentacyjny zazwyczaj krótko opisuje przeznaczenie i sposób użycia klasy (patrz listing 1.18). Przeanalizujmy kilka przykładów.

***Listing 1.18.*** *Jednowierszowy komentarz dokumentacyjny*

```
class Student:
    """Klasa implementująca operacje wykonywane przez studenta."""
    def __init__(self):
        pass
```

Powyższy jednowierszowy komentarz krótko opisuje klasę `Student`. Tworząc tego rodzaju komentarze, przestrzegaj opisanych wcześniej zasad.

Przeanalizujmy wielowierszowy komentarz do klasy przedstawiony w listingu 1.19.

***Listing 1.19.*** *Wielowierszowy komentarz do klasy*

```
class Student:
"""Informacje o klasie Student.

Klasa implementuje operacje wykonywane przez studenta. Zawiera m.in. informacje
o nazwisku, wieku i ocenach.
Użycie
import student
student = student.Student()
student.get_name()
>>> 678998
"""
def __init__(self):
    pass
```

Powyższy wielowierszowy komentarz zawiera nieco więcej informacji o przeznaczeniu klasy i sposobie jej użycia.

## Komentarze dokumentacyjne do funkcji

Komentarz dokumentacyjny umieszcza się na początku funkcji. Zazwyczaj opisuje on jej działanie. Jeżeli nie używasz modułu `typing`, opisuj w komentarzu również argumenty funkcji. Listing 1.20 przedstawia przykładowy komentarz.

***Listing 1.20.*** *Komentarz dokumentacyjny do funkcji*

```
def is_prime_number(number):
    """Sprawdzenie, czy liczba jest pierwsza.

    Funkcja sprawdza, czy wśród kolejnych liczb mniejszych od
    pierwiastka kwadratowego zadanej liczby znajduje się liczba pierwsza.

    :param number: Sprawdzana liczba.
    :type number: int
    :return: True, jeżeli liczba jest pierwsza, lub False w przeciwnym wypadku.
    :rtype: boolean
    """
...
```

## Przydatne narzędzia dokumentacyjne

Dostępnych jest bardzo dużo narzędzi tworzących na podstawie komentarzy dokumentację kodu w formacie HTML. Aby zaktualizować dokumentację, nie trzeba jej ręcznie zmieniać. Wystarczy jedynie użyć prostego polecenia. Narzędzia te stosowane w procesie tworzenia oprogramowania w dłuższym okresie są nieocenione. Każde z nich ma inne przeznaczenie.

- **Sphinx** (*http://www.sphinx-doc.org/en/stable*) — najpopularniejsze narzędzie automatycznie generujące dokumentację kodu w różnych formatach.
- **Pycco** (*https://pycco-docs.github.io/pycco*) — narzędzie do szybkiego generowania dokumentacji. Jego największą zaletą jest możliwość wyświetlania dokumentacji obok kodu.
- **Read the Docs** (*https://readthedocs.org*) — popularne, bezpłatne narzędzie umożliwiające tworzenie, wersjonowanie i przechowywanie dokumentacji.
- **Epydocs** (*http://epydoc.sourceforge.net*) — narzędzie do tworzenia dokumentacji modułów.

Dzięki powyższym narzędziom łatwiej jest w dłuższym okresie utrzymywać kod i dokumentację w spójnym formacie.

---

■ **Uwaga** Komentarze dokumentacyjne są doskonałą funkcjonalnością języka Python. Dzięki nim tworzenie dokumentacji kodu jest naprawdę proste, więc stosuj je jak najczęściej. Zaoszczędzisz wtedy mnóstwo czasu, szczególnie gdy projekt się rozwinie i będzie zawierał kilka tysięcy wierszy kodu.

---

# Pythoniczne struktury sterujące

Struktury sterujące stanowią fundamentalną część każdego języka programowania. Dotyczy to również Pythona, w którym struktury można kodować na różne sposoby. W tej części rozdziału opisano kilka dobrych praktyk, dzięki którym kod jest bardziej czytelny.

## Wyrażenia listowe

Wyrażenie listowe funkcjonuje podobnie jak pętla `for`. Można w nim stosować instrukcję `if`. W Pythonie istnieje kilka sposobów tworzenia listy na podstawie innej listy. Służą do tego celu głównie funkcje `filter()` i `map()`. Zalecane jest jednak stosowanie wyrażeń listowych, ponieważ dzięki nim kod jest o wiele bardziej czytelny.

Poniższy kod wylicza kwadraty liczb za pomocą funkcji `map()`:

```
numbers = [10, 45, 34, 89, 34, 23, 6]
square_numbers = map(lambda num: num**2, numbers)
```

Wersja wykorzystująca wyrażenie listowe wygląda tak:

```
square_numbers = [num**2 for num in numbers]
```

Przeanalizujmy inny przykład kodu filtrującego wartości `True`. Poniżej przedstawiona jest wersja wykorzystująca funkcję `filter()`:

```
data = [1, "A", 0, False, True]
filtered_data = filter(None, data)
```

Wersja wykorzystująca wyrażenie listowe wygląda następująco:

```
filtered_data = [item for item in data if item]
```

Jak widać, kod z wyrażeniami listowymi jest bardziej czytelny niż z funkcjami `filter()` i `map()`. Również oficjalna dokumentacja do Pythona zaleca stosowanie tych wyrażeń.

Jeżeli nie używasz skomplikowanych warunków lub pętli, stosuj wyrażenia listowe. Jednak jeżeli w wyrażeniu ma być wykonywanych kilka operacji, wtedy ze względu na czytelność kodu lepiej pozostać przy pętli.

Aby zilustrować przewagę wyrażenia nad pętlą, przeanalizujmy przykład kodu wyszukującego samogłoski w liście liter:

```
list_char = ["a", "p", "t", "i", "y", "l"]
vowel = ["a", "e", "i", "o", "u", "y"]
only_vowel = []
for item in list_char:
    if item in vowel:
        only_vowel.append(item)
```

Poniższa wersja wykorzystuje wyrażenie listowe:

```
only_vowel = [item for item in list_char if item in vowel]
```

Jak widać, wyrażenie listowe jest bardziej czytelne niż pętla i zajmuje mniej wierszy kodu. Ponadto jest bardziej wydajne, ponieważ nie jest w nim za każdym razem dopisywana wartość do listy, jak to ma miejsce w przypadku pętli. Wyrażenie jest również bardziej wydajne niż funkcje `filter()` i `map()`.

## Nie twórz skomplikowanych wyrażeń listowych

Wyrażenia listowe nie powinny być nadmiernie skomplikowane, ponieważ kod jest wtedy mniej czytelny i bardziej podatny na błędy. Jedno wyrażenie powinno zastępować najwyżej dwie pętle. Jeżeli będzie ich więcej, czytelność kodu pogorszy się.

Przeanalizujmy inny przykład użycia wyrażenia listowego. Zadanie polega na przetransponowaniu następującej tablicy:

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]
```

Wynikiem ma być poniższa tablica:

```
[[1,4,7],
 [2,5,8],
 [3,6,9]]
```

Wyrażenie listowe transponujące tablicę ma następującą postać:

```
return [[ matrix[row][col] for row in range(0, 3) ] for col in range(0, 3) ]
```

Powyższe wyrażenie jest czytelne, więc użycie go jest uzasadnione. Można je nawet zapisać w następującej postaci:

```
return [[ matrix[row][col]
        for row in range(0, 3) ]
        for col in range(0, 3) ]
```

Pętle zamiast wyrażeń listowych stosuj wtedy, gdy warunków jest kilka, jak niżej:

```
ages = [1, 34, 5, 7, 3, 57, 356]
old = [age for age in ages if age > 10 and age < 100 and age is not None]
```

W drugim wierszu dużo się dzieje, przez co kod jest mniej czytelny i podatny na błędy. Dlatego lepszym rozwiązaniem jest zastosowanie pętli. Kod może wyglądać na przykład tak:

```
ages = [1, 34, 5, 7, 3, 57, 356]
old = []
for age in ages:
    if age > 10 and age < 100:
        old.append(age)
```

W tej wersji kod składa się z większej liczby wierszy, ale jest bardziej czytelny.

Niepisaną dobrą praktyką jest tworzenie najpierw wyrażeń listowych, a następnie zamienianie ich na pętle, jeżeli wyrażenia okażą się zbyt skomplikowane i nieczytelne.

---

■ **Uwaga** Rozważne stosowanie wyrażeń listowych bardzo poprawia jakość kodu. Jednak jeżeli używa się ich nadmiernie, kod przestaje być czytelny. Dlatego lepiej jest z nich zrezygnować, jeżeli robią się skomplikowane, np. zawierają więcej niż dwa warunki lub dwie pętle.

---

## Kiedy stosować wyrażenia lambda?

Wyrażenia lambda warto stosować zamiast funkcji wtedy, gdy dzięki nim kod staje się bardziej czytelny. Przeanalizujmy przykład pokazany w listingu 1.21.

***Listing 1.21.*** *Wyrażenie lambda bez instrukcji przypisania*

```
data = [[7], [3], [0], [8], [1], [4]]
def min_val(data):
    """Wyszukiwanie minimalnej wartości w liście."""
    return min(data, key=lambda x: len(x))
```

W powyższym kodzie wyrażenie lambda zostało użyte w charakterze doraźnej funkcji wyszukującej minimalną wartość w liście. Jednak lepiej nie stosować wyrażenia lambda jako anonimowej funkcji, jak niżej:

```
min_val = min(data, key=lambda x: len(x))
```

Tutaj wartość zmiennej `min_val` jest wyliczana za pomocą wyrażenia lambda. Stosowanie wyrażenia lambda zamiast funkcji jest powieleniem instrukcji `def`, co nie jest zgodne z filozofią Pythona, która zaleca, aby każdą rzecz robić tylko w jeden sposób. Dokumentacja PEP8 tak mówi na ten temat:

> *W instrukcjach przypisujących wartości bezpośrednio do zmiennych zawsze stosuj funkcje, a nie wyrażenia lambda.*
>
> *Rób tak:*
>
> `def f(x): return 2*x`
>
> *A nie tak:*
>
> `f = lambda x: 2*x`
>
> *W pierwszym przypadku obiekt ma konkretną nazwę* `f`*, a nie ogólną* `lambda`*. Dzięki temu komunikaty o błędach są bardziej czytelne, jak również łatwiej uzyskać tekstową reprezentację obiektu. W instrukcji przypisania nie jest wykorzystywana najważniejsza cecha wyrażenia lambda, tj. możliwość osadzania go w większych wyrażeniach.*

## Kiedy stosować generatory, a kiedy wyrażenia listowe?

Wyrażenie listowe różni się od generatora przede wszystkim tym, że zajmuje pamięć. Dlatego wyrażenia należy stosować w następujących sytuacjach:

- gdy lista wartości jest iterowana wielokrotnie,
- gdy dane są przetwarzane za pomocą metod niedostępnych w generatorach,
- gdy danych nie jest dużo i umieszczenie ich w pamięci nie będzie przyczyną problemów.

Przeanalizujmy kod przedstawiony w listingu 1.22, wyszukujący w pliku określone wiersze.

***Listing 1.22.*** *Odczytywanie wierszy z pliku za pomocą wyrażenia listowego*

```
def read_file(file_name):
    """Odczytywanie pliku wiersz po wierszu."""
    fread = open(file_name, "r")
    data = [line for line in fread if line.startswith(">>")]
    return data
print(read_file("log.txt"))
```

Jeżeli plik jest duży, wtedy umieszczenie wierszy w liście może skutkować zajęciem dużej ilości pamięci i spowolnieniem działania kodu. Dlatego w takim przypadku lepiej jest stosować generator, jak w listingu 1.23.

***Listing 1.23.*** *Odczytywanie wierszy z pliku za pomocą iteratora*

```
def read_file(file_name):
    """Odczytywanie pliku wiersz po wierszu."""
    with open(file_name) as fread:
        for line in fread:
            yield line
for line in read_file("log.txt"):
    if line.startswith(">>"):
        print(line)
```

W powyższym listingu dane nie są umieszczane w pamięci w formie wyrażenia listowego, tylko odczytywane i przetwarzane wiersz po wierszu. Wyrażenie jednak można wykorzystywać wielokrotnie, natomiast generator trzeba uruchamiać za każdym razem, gdy zaistnieje konieczność przetworzenia wierszy zaczynających się od znaków >>. Obie funkcjonalności są jednak bardzo ważne i stosując je zgodnie z powyższym opisem, można tworzyć wydajny kod.

## Dlaczego nie należy stosować instrukcji else w pętlach?

W Pythonie instrukcję `else` można stosować w pętlach `for` i `while`. Kod umieszczony w bloku `else` jest wykonywany po zakończeniu działania pętli. Jeżeli pętla zostanie przerwana za pomocą instrukcji `break`, wtedy blok `else` nie jest wykonywany.

Instrukcja `else` użyta w pętli może wprowadzać zamieszanie w kodzie, dlatego wielu programistów jej nie stosuje. Jest to zrozumiałe, gdy porówna się działanie tej instrukcji w zwykłej strukturze `if/else`.

Przeanalizujmy prosty przykład przedstawiony w listingu 1.24. Jest to pętla przetwarzająca listę, zawierająca instrukcję `else`.

***Listing 1.24.*** *Pętla z instrukcją else*

```
for item in [1, 2, 3]:
    print("Iteracja")
else:
    print("Koniec")
```

Wynik:

```
Iteracja
Iteracja
Iteracja
Koniec
```

Wydawałoby się, że blok `else` powinien być pominięty, a wynik powinien zawierać tylko słowa `Iteracja`. Tak nakazuje logika właściwa dla struktury `if/else`. Jednakże w przypadku pętli jest inaczej. Rzecz się komplikuje jeszcze bardziej w pętli `while`, jak w listingu 1.25.

***Listing 1.25.*** *Pętla while z instrukcją else*

```
x = [1, 2, 3]
while x:
    print("Iteracja")
    x.pop()
else:
    print("Koniec")
```

Wynik działania kodu jest następujący:

```
Iteracja
Iteracja
Iteracja
Koniec
```

W tym przypadku pętla jest wykonywana do chwili opróżnienia listy, po czym jest wykonywany blok `else`.

Opisywana funkcjonalność ma swoje uzasadnienie. Jednym z jej zastosowań jest implementacja dodatkowych operacji wykonywanych tylko po normalnym zakończeniu działania pętli (patrz listing 1.26).

***Listing 1.26.*** *Pętla z instrukcjami break i else*

```
for item in [1, 2, 3]:
    if item % 2 == 0:
        break
    print("Iteracja")
else:
    print("Koniec")
```

Wynik jest następujący:

```
Iteracja
```

Są jednak lepsze sposoby kodowania niż implementacja pętli z instrukcją `break` w środku (lub bez niej) oraz `else` na końcu. Ten sam efekt można osiągnąć bez instrukcji `else`. Należy użyć instrukcji `if` (patrz listing 1.27). Kod na pierwszy rzut oka będzie mniej czytelny, ale ryzyko niezrozumienia go przez innego programistę jest mniejsze.

***Listing 1.27.*** *Pętla z instrukcją break*

```
flag = True
for item in [1, 2, 3]:
    if item % 2 == 0:
        flag = False
        break
    print("Iteracja")
if flag:
    print("Koniec")
```

Wynik działania kodu jest następujący:

```
Koniec
```

Powyższy kod jest łatwiej zrozumieć i nie ma możliwości jego błędnej interpretacji.

Instrukcja `else` jest ciekawą funkcjonalnością języka, jednak może pogarszać czytelność kodu. Dlatego lepiej unikać jej stosowania.

## Dlaczego warto stosować funkcję range() w języku Python 3?

W języku Python 2 na pewno stosowałeś funkcję `xrange()`. W wersji 3 pojawiła się funkcja `range()`, podobna do swojej poprzedniczki, ale iterowalna i oferująca kilka dodatkowych możliwości.

```
>>> range(4)
range(0, 5)          # Iterowalny zakres
>>> list(range(4))
[0, 1, 2, 3, 4]      # Lista
```

Przewaga funkcji `range()` nad listą polega przede wszystkim na tym, że dane nie są umieszczane w pamięci. W odróżnieniu od list, krotek i innych struktur, funkcja `range()` reprezentuje niemutowalny, iterowalny obiekt, zajmujący niewielką ilość pamięci niezależnie od rozmiaru danych. Przechowywane są jedynie wartości początkowa, końcowa i przyrostowa. Dane są zawsze generowane na żądanie.

Za pomocą funkcji `range()` można wykonywać kilka opisanych niżej operacji nieosiągalnych za pomocą `xrange()`.

- Porównywanie zakresów danych:

```
>>> range(4) == range(4)
True
>>> range(4) == range(5)
False
```

- Wyodrębnianie fragmentów danych:

```
>>> range(10)[2:]
range(2, 10)
>>> range(10)[2:7, -1)
range(2, 7, -1)
```

Więcej informacji o funkcji `range()` znajdziesz na stronie *https://docs.python.org/3.0/library/functions.html#range*.

Jeżeli trzeba wykonywać operacje na liczbach, lepiej jest stosować funkcję `range()`, a nie listę, ponieważ funkcja ta jest wydajniejsza, jak również łatwiej ją stosować w pętli.

**Nie rób tak:**

```
for item in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(item)
```

**Tylko tak:**

```
for item in range(10):
    print(item)
```

Wydajność pierwszej pętli jest znacznie mniejsza niż drugiej, ponieważ każda wartość musi być odczytana z pamięci. Jeżeli zakres wartości jest duży, wtedy kod będzie działał znacznie wolniej.

# Zgłaszanie wyjątków

Wyjątki są symptomami błędów w kodzie. W Pythonie wyjątki obsługuje wbudowany moduł. Dogłębna znajomość wyjątków, miejsc i okoliczności, w których są zgłaszane, pozwala tworzyć kod mniej podatny na błędy.

Za pomocą wyjątków można łatwo sygnalizować błędy w kodzie, dlatego warto z nich korzystać. Pozwalają poznać ograniczenia wykorzystywanego interfejsu API lub biblioteki. Zgłaszane w odpowiednim miejscu ułatwiają innym programistom korzystanie z kodu lub interfejsu API.

## Często zgłaszane wyjątki

Zapewne zastanawiasz się, gdzie i kiedy należy zgłaszać wyjątki. Według mnie trzeba to robić wtedy, gdy nie są spełniane podstawowe założenia przyjęte w danym bloku kodu. Preferowane jest zgłaszanie wyjątków w przypadku pojawienia się poważnego błędu. Należy je zgłaszać nawet wtedy, gdy pojawia się permanentny problem.

Przeanalizujmy listing 1.28, w którym jedna liczba jest dzielona przez inną.

***Listing 1.28.*** *Dzielenie liczb i zgłaszanie wyjątku*

```
def division(dividend, divisor):
    """Dzielenie dwóch liczb."""
    try:
        return dividend/divisor
    except ZeroDivisionError as zero:
        raise ZeroDivisionError("Dzielnik musi być różny od zera")
```

Jak pokazuje powyższy kod, wyjątki należy zgłaszać w tych miejscach kodu, w których pojawiają się błędy. Dzięki temu w przypadku wystąpienia problemu zewnętrzny kod będzie mógł odpowiednio zareagować (patrz listing 1.29).

***Listing 1.29.*** *Dzielenie liczb bez zgłaszania wyjątku*

```
result = division(10, 2)
```

Co się stanie, gdy funkcja nie zgłosi wyjątku, tylko zwróci wynik `None`?

```
def division(dividend, divisor):
    """Dzielenie dwóch liczb."""
    try:
        return dividend/divisor
    except ZeroDivisionError as zero:
        return None
```

Powyższa funkcja w przypadku wystąpienia wyjątku `ZeroDivisionError` nie zgłosi problemu, tylko zwróci wynik `None`. Z tego powodu trudniej będzie diagnozować wywołujący ją kod, szczególnie jeżeli będzie on rozbudowany lub zmienią się stawiane mu wymagania. Funkcja nie powinna zwracać wyniku `None`, ponieważ w wywołującym ją kodzie nie będzie można określić przyczyny problemu. Jeżeli natomiast będzie zgłaszany wyjątek, wtedy w kodzie wywołującym funkcję będzie od razu wiadomo, że wartości argumentów są błędne i trzeba poprawić ukryte błędy. W zewnętrznym kodzie łatwiej obsługiwać wyjątki niż wyniki funkcji sygnalizujące problem.

Zasada obowiązująca w Pythonie brzmi: lepiej prosić o wybaczenie niż o pozwolenie. Oznacza to, że nie należy sprawdzać zawczasu, czy pojawi się błąd, tylko obsługiwać wyjątek, gdy zostanie już zgłoszony. Zatem należy w kodzie zgłaszać wyjątki wszędzie tam, gdzie mogą występować błędy. Zewnętrzny kod nie będzie wtedy „zaskakiwany” i będzie mógł odpowiednio reagować.

## Obsługuj wyjątki za pomocą instrukcji finally

Kod w bloku `finally` jest wykonywany niezależnie od tego, czy został zgłoszony wyjątek, czy nie. Instrukcja ta przydaje się szczególnie w sytuacjach, gdy kod wykorzystuje zewnętrzne zasoby. Dzięki niej można np. zamykać pliki i zwalniać zasoby. Przeanalizujmy kod w listingu 1.30.

***Listing 1.30.*** *Przykład użycia instrukcji finally*

```
def send_email(host, port, user, password, email, message):
    """Wysłanie wiadomości e-mail na zadany adres."""
    try:
        server = smtlib.SMTP(host=host, port=port)
        server.ehlo()
        server.login(user, password)
        server.send_email(message)
    finally:
        server.quit()
```

W tej sytuacji w bloku `finally` w przypadku zgłoszenia wyjątku przez metodę `login()` lub `send_mail()` zwalniane są zasoby zajmowane podczas nawiązywania połączenia z serwerem.

W bloku `finally` można zamykać otwarte pliki, jak w listingu 1.31.

***Listing 1.31.*** *Zamykanie plików w bloku finally*

```
def write_file(file_name):
    """Zapisanie wiersza w pliku."""
    myfile = open(file_name, "w")
    try:
        myfile.write("Python jest świetny!")    # Możliwy wyjątek.
    finally:
        myfile.close()      # Instrukcja wykonywana przed eskalacją błędu.
```

Jeżeli więc jakiś kod ma być wykonywany niezależnie od tego, czy zostanie zgłoszony wyjątek, czy nie, należy go umieszczać w bloku `finally`. Dzięki temu zasoby będą właściwie wykorzystywane, a kod bardziej czytelny.

## Twórz własne klasy wyjątków

Tworząc interfejs API lub bibliotekę, warto definiować własne klasy wyjątków, dzięki którym łatwiej będzie diagnozować przyczyny problemów.

Załóżmy, że wyjątek ma być zgłaszany w przypadku, gdy w bazie danych nie zostaną znalezione dane o użytkowniku. Klasa wyjątku musi opisywać problem. Nazwa wyjątku, np. `UserNotFoundError`, informuje o przyczynie zgłoszenia wyjątku.

Listing 1.32 przedstawia przykład klasy wyjątku zaimplementowanej w języku Python 3.

***Listing 1.32.*** *Własna klasa wyjątku*

```
class UserNotFoundError(Exception):
    """Zgłoszenie wyjątku, gdy użytkownik nie zostanie znaleziony."""
    def __init__(self, message=None, errors=None):
        # Wywołanie konstruktora klasy bazowej z wymaganymi argumentami.
        super().__init__(message)
        self.errors = errors

def get_user_info(user_obj):
    """Odczytanie danych użytkownika z bazy."""
    user = get_user_from_db(user_obj)
    if not user:
        raise UserNotFoundException(f"Brak użytkownika o identyfikatorze
{user_obj.id}")

get_user_info(user_obj)
>>> UserNotFoundException: Brak użytkownika o identyfikatorze 897867
```

Własna klasa wyjątku powinna być opisowa i mieć jasno określone przeznaczenie. Wyjątek `UserNotFoundException` powinien być zgłaszany tylko w tych miejscach, w których nie można z bazy odczytać informacji o użytkowniku. Jeżeli klasa ma jasno określone granice, wtedy łatwiej jest diagnozować kod i wiadomo, co było przyczyną zgłoszenia wyjątku.

Klasa wyjątku może obejmować szerszy zakres przypadków, ale musi mieć wtedy odpowiednią nazwę, jak w listingu 1.33. Tutaj nazwa `ValidationError` opisuje różne przypadki weryfikacji danych i obejmuje wszystkie związane z nią wyjątki.

***Listing 1.33.*** *Własna klasa wyjątku o szerszym zakresie*

```
class ValidationError(Exception):
    """Zgłoszenie wyjątku w przypadku błędu weryfikacji danych."""
    def __init__(self, message=None, errors=None):
        # Wywołanie konstruktora klasy bazowej z wymaganymi argumentami.
        super().__init__(message)
        self.errors = errors
```

Zakres tej klasy jest znacznie szerszy niż klasy `UserNotFoundException`. Wyjątek `ValidationError` można zgłaszać wszędzie tam, gdzie może wystąpić błąd weryfikacji danych lub dane mogą mieć niewłaściwy format. Upewniaj się więc, że klasa obejmuje właściwy zakres i wyjątki są zgłaszane w przypadku wystąpienia właściwego dla niej błędu.

## Obsługuj konkretne wyjątki

Zamiast stosować instrukcję `except` obsługującą wszystkie wyjątki, lepiej jest implementować obsługę konkretnych wyjątków. Instrukcja `except` obejmuje wszystkie rodzaje wyjątków, przez co krytyczne błędy w kodzie mogą pozostać niezauważone albo kod obsługujący wyjątki może nie działać zgodnie z przeznaczeniem.

Przeanalizujmy poniższy fragment, w którym w funkcji `get_even_list` zastosowana jest instrukcja `except`.

**Nie rób tak:**

```
def get_even_list(num_list):
    """Wybranie z listy liczb parzystych."""
    # Tu może być zgłoszony wyjątek RuntimeError lub TypeError.
    return [item for item in num_list if item%2==0]

numbers = None
try:
    get_even_list(numbers)
except:
    print("Coś poszło nie tak.")
>>> Coś poszło nie tak.
```

W tym kodzie nie są rozróżniane wyjątki `RuntimeError` i `TypeError`, co jest oczywistym błędem, ponieważ w nadrzędnym kodzie nie będzie można na podstawie komunikatu `Coś poszło nie tak.` określić przyczyny problemu. Gdyby był zgłaszany wyjątek z odpowiednim komunikatem, wtedy jego obsługa w nadrzędnym kodzie byłaby znacznie łatwiejsza.

Instrukcja `except` użyta bez argumentu dotyczy wyjątku `BaseException`. Jawne wskazanie wyjątku ułatwia diagnostykę, szczególnie dłuższego kodu.

**Rób tak:**

```
def get_even_list(num_list):
    """Wybranie z listy liczb parzystych."""
    # Tu może być zgłoszony wyjątek RuntimeError lub TypeError.
    return [item for item in num_list if item%2==0]

numbers = None
try:
    get_even_list(numbers)
except TypeError:
    print("Lista może zawierać wyłącznie liczby.")
except RuntimeError:
    print("Błąd wykonania kodu.")
```

Obsługa konkretnych wyjątków ułatwia diagnostykę. W kodzie nadrzędnym od razu wiadomo, co jest przyczyną problemu i co można zrobić, aby obsłużyć dany wyjątek. Dzięki temu kod staje się mniej podatny na błędy.

Zgodnie z dokumentacją PEP8 należy używać instrukcji `except` w następujących przypadkach:

- Gdy moduł obsługi wyjątków wyświetla lub loguje informacje diagnostyczne. Wtedy przynajmniej użytkownik jest powiadamiany, że pojawił się błąd.
- Gdy kod musi wykonać pewne operacje porządkujące, a następnie wyjątek musi zostać eskalowany do kodu nadrzędnego za pomocą instrukcji `raise`. Można wtedy stosować konstrukcję `try/except`.

---

■ **Uwaga** Obsługiwanie konkretnych wyjątków jest jedną z najlepszych praktyk pisania kodu, szczególnie w języku Python, ponieważ oszczędza się wtedy mnóstwo czasu na diagnostyce. W przypadku pojawienia się błędu kod natychmiast ulega awarii, zamiast ukrywać błąd.

---

## Zwracaj uwagę na zewnętrzne wyjątki

Podczas korzystania z zewnętrznego interfejsu API bardzo ważna jest znajomość wszystkich zgłaszanych przez niego wyjątków. Dzięki temu łatwiej jest diagnozować problemy.

Jeżeli uważasz, że dany wyjątek nie odzwierciedla określonego błędu, utwórz własną klasę wyjątku. Gdy korzystasz z zewnętrznych bibliotek, twórz własne klasy, aby nadawać wyjątkom nazwy odpowiednie do potencjalnych błędów lub definiować komunikaty zwracane w przypadku zgłoszenia zewnętrznego wyjątku.

Przeanalizujmy kod przedstawiony w listingu 1.34 wykorzystujący bibliotekę `botocore`.

***Listing 1.34.*** *Obsługa zewnętrznego wyjątku*

```
from botocore.exceptions import ClientError

ec2 = session.get_client('ec2', 'us-east-2')
try:
    parsed = ec2.describe_instances(InstanceIds=['i-badid'])
except ClientError as e:
    logger.error("Błąd: %s", e, exc_info=True)
    # Obsługa tylko określonego przypadku błędu.
    if e.response['Error']['Code'] == 'InvalidInstanceID.NotFound':
        raise WrongInstanceIDError(message=exc_info, errors=e)

class WrongInstanceIDError(Exception):
    """Zgłoszenie wyjątku w przypadku błędnej instancji."""
    def __init__(self, message=None, errors=None):
        # Wywołanie konstruktora klasy bazowej z wymaganymi argumentami.
        super().__init__(message)
        self.errors = errors
```

Zwróć uwagę na dwie rzeczy w powyższym kodzie:

- Dzięki obsłudze konkretnego wyjątku łatwiej jest diagnozować błędy zgłaszane przez zewnętrzną bibliotekę.
- Zdefiniowana jest nowa klasa wyjątku. Nie trzeba tego robić dla każdego wyjątku, ale dzięki temu kod jest bardziej czytelny.

Czasami trudno jest znaleźć odpowiedni sposób obsługi wyjątków zgłaszanych przez zewnętrzną bibliotekę lub interfejs API. Znajomość przynajmniej najczęściej zgłaszanych wyjątków ułatwia diagnozowanie błędów w kodzie produkcyjnym.

## Twórz jak najmniejsze bloki try

W kodzie obsługującym wyjątki twórz jak najmniejsze bloki `try`. Dzięki temu inni programiści będą wiedzieć, jakie błędy mogą pojawić się w danej części kodu. Ponadto im mniejszy jest blok `try`, tym łatwiej jest diagnozować kod. Program, w którym nie są stosowane konstrukcje `try/except`, działa nieco szybciej, ale w przypadku pojawienia się błędu ulega awarii. Dobrze zaimplementowana obsługa wyjątków uodparnia kod na błędy i pozwala zaoszczędzić mnóstwo czasu na diagnozowaniu go w środowisku produkcyjnym.

Przeanalizujmy poniższy kod.

```
def write_to_file(file_name, message):
    """Zapisanie komunikatu w pliku."""
    try:
        write_file = open(file_name, "w")
        write_file.write(message)
        write_file.close()
    except FileNotFoundError as exc:
        print("Podaj poprawną nazwę pliku")
```

Gdy przyjrzysz mu się bliżej, zauważysz, że może zgłaszać przynajmniej dwa wyjątki: `FileNotFoundError` i `IOError`. Oba można obsłużyć w jednym wierszu lub w osobnych blokach `try`.

**Rób tak:**

```
def write_to_file(file_name, message):
    """Zapisanie komunikatu w pliku."""
    try:
        write_file = open(file_name, "w")
        write_file.write(message)
        write_file.close()
    except (FileNotFoundError, IOError) as exc:
        print(f"Błąd zapisu do pliku {exc}")
```

Nawet jeżeli nie ma ryzyka pojawienia się wyjątku, zaleca się tworzenie minimalnego bloku `try`, jak niżej.

**Nie rób tak:**

```
try:
    data = get_data_from_db(obj)
    return data
except DBConnectionError:
    raise
```

**Tylko tak:**

```
try:
    data = get_data_from_db(obj)
except DBConnectionError:
    raise
return data
```

Dzięki temu kod jest bardziej czytelny i wiadomo, że wyjątek może zostać zgłoszony tylko przez funkcję `get_data_from_db()`.

# Podsumowanie

W tym rozdziale poznałeś kilka popularnych praktyk, dzięki którym kod staje się prostszy i bardziej czytelny.

Obsługa wyjątków jest jednym z najważniejszych aspektów kodowania w języku Python. Dogłębna znajomość wyjątków ułatwia diagnozowanie aplikacji. Dotyczy to szczególnie dużych projektów produkcyjnych, składających się z wielu części tworzonych przez różnych programistów. Zgłaszanie wyjątków w odpowiednich miejscach pozwala zaoszczędzić mnóstwo czasu i pieniędzy, szczególnie podczas diagnozowania kodu produkcyjnego. Logowanie i obsługa wyjątków to cechy dojrzałej aplikacji, dlatego należy zawczasu o nich pomyśleć, aby kod był bardziej czytelny i łatwiejszy w utrzymaniu.

# Python: kodowanie jest sztuką!

W porównaniu z innymi językami programowania Python wyróżnia się prostotą i zaskakującymi możliwościami. Używa się go do analizy danych, budowania sztucznej inteligencji, tworzenia stron WWW, jak również w badaniach naukowych. Właściwości tego języka sprawiają, że kod trzeba pisać uważnie, szczególnie w dużych projektach. Tymczasem trudno jest znaleźć odpowiednio dobre źródło informacji o sposobach tworzenia kodu wysokiej jakości. Publikacje dotyczące dobrych praktyk kodowania w Pythonie są nieliczne, a ich jakość nieraz pozostawia wiele do życzenia. Dodatkowy problem wynika z wszechstronności Pythona: jest wykorzystywany w wielu dziedzinach i trudno wskazać wspólne dla nich wzorce programistyczne.

Ta książka jest znakomitym przewodnikiem, dzięki któremu będziesz tworzyć wydajne i bezbłędne aplikacje w Pythonie. Zaczniesz od formatowania i dokumentowania kodu, umiejętnego stosowania wbudowanych struktur, stosowania modułów i metaklas. W ten sposób nauczysz się pisać uporządkowany kod. Potem poznasz nowe funkcjonalności języka Python i dowiesz się, jak efektywnie z nich korzystać. Następnie dowiesz się, jak wykorzystywać zaawansowane cechy języka, takie jak programowanie asynchroniczne, określanie typów danych i obsługa ścieżek, a także jak diagnozować kod, wykonywać testy jednostkowe i integracyjne oraz przygotowywać kod do uruchomienia w środowisku produkcyjnym. Na końcu poznasz najważniejsze narzędzia przeznaczone do szybkiego tworzenia kodu, zarządzania jego wersjami i weryfikowania poprawności.

W tej książce między innymi:

- właściwe wykorzystywanie wyrażeń i instrukcji
- tworzenie własnych słowników
- zaawansowane struktury danych
- pisanie najlepszych modułów, klas i funkcji
- asynchroniczne wywoływanie funkcji

**Sunil Kapil** od dziesięciu lat tworzy aplikacje produkcyjne w Pythonie i kilku innych językach. Zajmuje się głównie oprogramowaniem dla usług internetowych i mobilnych. Jest pasjonatem otwartego oprogramowania i aktywnie uczestniczy w projektach open source. Ponadto działa w organizacjach non profit, w których nieodpłatnie zajmuje się projektami informatycznymi. Często występuje jako prelegent na konferencjach, zazwyczaj mówi wtedy o Pythonie. Prowadzi też stronę o inżynierii oprogramowania, przydatnych narzędziach i technikach.

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





ISBN 978-83-283-6462-2



Cena: 39,90 zł

Apress