Sprawdzone przepisy dla programistów Pythona!

Wydanie III

Python Receptury

HELION

O'REILLY®

David Beazley, Brian K. Jones

Tytuł oryginału: Python Cookbook, 3rd Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-246-8180-8

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pytre3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pytre3.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 14.

Testowanie, debugowanie i wyjątki

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

14.1. Testowanie danych wyjściowych wysyłanych do strumienia stdout

Problem

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

Rozwiązanie

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

Przyjrzyj się przykładowej funkcji z modułu `mymodule`:

```
# mymodule.py

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

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

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

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

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

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

Omówienie

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

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

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

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

14.2. Podstawianie obiektów w testach jednostkowych

Problem

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

Rozwiązanie

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

```
from unittest.mock import patch
import example

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

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

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

Ponadto można ją zastosować do ręcznego podstawiania elementów kodu:

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

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

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

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

Omówienie

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

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

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

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

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

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

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

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

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

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

def dowprices():
   u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=sl1')
   lines = (line.decode('utf-8') for line in u)
   rows = (row for row in csv.reader(lines) if len(row) == 2)
   prices = { name:float(price) for name, price in rows }
   return prices
```

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

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

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

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

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

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

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

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

14.3. Sprawdzanie wystąpienia wyjątków w testach jednostkowych

Problem

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

Rozwiązanie

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

```
import unittest

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

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

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

```
import errno

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

Omówienie

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

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

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

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

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

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

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

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

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

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

14.4. Zapisywanie danych wyjściowych testu w pliku

Problem

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

Rozwiązanie

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

```
import unittest

class MyTest(unittest.TestCase):
   ...

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

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

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

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

Omówienie

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

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

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

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

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

14.5. Pomijanie testów lub przewidywanie ich niepowodzenia

Problem

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

Rozwiązanie

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

```
import unittest
import os
import platform

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

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

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

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

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

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

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

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

OK (skipped=2, expected failures=1)
```

Omówienie

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

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

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

14.6. Obsługa wielu wyjątków

Problem

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

Rozwiązanie

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

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

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

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

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

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

można zastosować polecenie `except` w poniższy sposób:

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

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

Omówienie

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

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

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

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

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

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

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

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

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

14.7. Przechwytywanie wszystkich wyjątków

Problem

Programista chce napisać kod, który przechwytuje wszystkie wyjątki.

Rozwiązanie

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

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

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

Omówienie

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

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

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

Jeśli uruchomisz tę funkcję, uzyskasz następujące dane:

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

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

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

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

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

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

14.8. Tworzenie niestandardowych wyjątków

Problem

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

Rozwiązanie

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

```
class NetworkError(Exception):
   pass

class HostnameError(NetworkError):
   pass

class TimeoutError(NetworkError):
   pass

class ProtocolError(NetworkError):
   pass
```

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

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

Omówienie

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

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

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

a także bardziej ogólne grupy błędów:

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

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

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

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

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

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

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

14.9. Zgłaszanie wyjątku w odpowiedzi na wystąpienie innego wyjątku

Problem

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

Rozwiązanie

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

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

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

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

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

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

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

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

During handling of the above exception, another exception occurred:

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

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

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

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

Omówienie

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

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

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

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

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

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

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

14.10. Ponowne zgłaszanie ostatniego wyjątku

Problem

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

Rozwiązanie

Wystarczy zastosować samo polecenie `raise`. Oto przykład:

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

Omówienie

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

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

   # Przekazywanie wyjątku
   raise
```

14.11. Wyświetlanie komunikatów ostrzegawczych

Problem

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

Rozwiązanie

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

```
import warnings

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

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

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

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

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

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

Omówienie

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

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

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

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

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

14.12. Debugowanie prostych awarii programu

Problem

Program nie działa i programista szuka prostej techniki debugowania.

Rozwiązanie

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

```
# sample.py

def func(n):
   return n + 10

func('Witaj')
```

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

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

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

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

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

```
import traceback
import sys

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

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

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

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

```
import pdb

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

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

Omówienie

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

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

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

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

14.13. Profilowanie i pomiar czasu pracy programów

Problem

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

Rozwiązanie

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

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

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

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

  Ordered by: standard name

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

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

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

```
# timethis.py

import time
from functools import wraps

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

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

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

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

```
from contextlib import contextmanager

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

Oto przykład ilustrujący działanie tego menedżera kontekstu:

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

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

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

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

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

Omówienie

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

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

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

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

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

14.14. Przyspieszanie działania programów

Problem

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

Rozwiązanie

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

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

Stosowanie funkcji

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

```
# somescript.py

import sys
import csv

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

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

```
# somescript.py
import sys
import csv

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

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

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

Wybiórcze eliminowanie operacji dostępu do atrybutów

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

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

```
import math

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

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

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

```
from math import sqrt

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

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

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

Zmienne lokalne

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

```
import math

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

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

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

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

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

Unikanie niepotrzebnej abstrakcji

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

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

Teraz przeprowadź prosty pomiar czasu:

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

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

Stosowanie wbudowanych kontenerów

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

Unikanie tworzenia niepotrzebnych struktur danych i kopii

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

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

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

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

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

Omówienie

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

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

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

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

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

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

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

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

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

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ż

Programowanie

O'REILLY®

Python. Receptury

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

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

Dzięki tej książce:

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

Najlepsze rozwiązania typowych problemów!

helion.pl
księgarnia internetowa

Nr katalogowy: 17110

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-8180-8

Cena 99,00 zł

9 788324 681808