Technologia i rozwiązania

Profesjonalne programowanie w Pythonie

Poziom ekspert

Wydanie II

Michał Jaworski
Tarek Ziadé

Helion

[PACKT] PUBLISHING open source* community experience distilled

Tytuł oryginału: Expert Python Programming, Second Edition

Tłumaczenie: Michał Jaworski

ISBN: 978-83-283-3033-7

First published in the English language under the title 'Expert Python Programming - Second Edition' – 9781785886850.

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)

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

Spis treści

ROZDZIAŁ 7

Rozszerzenia Pythona w innych językach programowania

Podczas pisania aplikacji bazujących na Pythonie nie musisz ograniczać się tylko do samego języka Python. Jedną z ciekawych alternatyw — język Hy — przedstawiłem w rozdziale 3., „Najlepsze praktyki składniowe — powyżej poziomu klas". Pozwala on na pisanie modułów, pakietów, a nawet całych aplikacji z wykorzystaniem innego języka (dialektu Lispa) działającego w wirtualnej maszynie Pythona. W kwestii dostarczanych możliwości powinien być traktowany na równi z Pythonem (pomimo swojej całkowicie odmiennej składni), ponieważ kompiluje się do tego samego kodu bajtowego. Niestety, oznacza to, że kod napisany w Hy posiada te same ograniczenia co zwykły kod napisany w Pythonie:

- użyteczność wątków jest znacznie ograniczona z powodu istnienia mechanizmu GIL;
- nie jest to język kompilowany;
- nie dostarcza mechanizmu statycznego typowania oraz wiążących się z nim potencjalnych optymalizacji.

Jednym z rozwiązań powyższych ograniczeń są rozszerzenia napisane w całkowicie innych językach programowania, które udostępniają swoje interfejsy za pomocą API rozszerzeń Pythona.

W tym rozdziale omówię podstawowe powody do pisania własnych rozszerzeń Pythona w innych językach programowania oraz popularne narzędzia mające na celu ułatwienie procesu ich tworzenia. Dowiesz się:

- jak napisać proste rozszerzenie w języku C z wykorzystaniem API Python/C;
- jak zrobić to samo z wykorzystaniem Cythona;

- jakie są największe wyzwania związane z pisaniem rozszerzeń oraz generowane przez nie problemy;
- jak korzystać z kompilowanych bibliotek dynamicznych bez potrzeby tworzenia dedykowanych rozszerzeń z wykorzystaniem surowego kodu Pythona.

Inne języki, czyli C lub C++

Kiedy mówię o innych językach (w kontekście rozszerzeń Pythona), niemal zawsze mam na myśli wyłącznie C i C++. Nawet narzędzia typu Cython i Pyrex, udostępniające nadzbiory języka Python na potrzeby budowania rozszerzeń, są tak naprawdę kompilatorami typu *źródła-do-źródeł*, które generują kod C na podstawie rozszerzonej składni Pythona.

Jest oczywiście możliwe ładowanie w Pythonie dynamicznych/współdzielonych bibliotek. Oznacza to, że w Pythonie można wykorzystać kod dowolnego innego języka, który umożliwia kompilację tego typu bibliotek. Jednakże biblioteki dynamiczne i współdzielone są z definicji uniwersalne. Można ich używać w kodzie każdego języka, który pozwala na ich ładowanie. Oznacza to, że za rozszerzenia Pythona można uznać tylko te biblioteki, które korzystają bezpośrednio z interfejsu Python/C.

Niestety, pisanie własnych rozszerzeń w języku C bądź C++ z wykorzystaniem surowego API Python/C jest zadaniem dosyć wymagającym, i to nie tylko z powodu konieczności posiadania szerokiej wiedzy na temat przynajmniej jednego z dwóch języków uważanych generalnie za trudne do opanowania. Otóż stworzenie nawet prostego rozszerzenia wymaga napisania ogromnej ilości elementów stałych. Są to powtarzające się fragmenty kodu, które mają na celu powiązanie Twojej zaimplementowanej logiki z Pythonem i jego wbudowanymi typami danych. Mimo wszystko dobrze jest znać typową budowę rozszerzeń Pythona napisanych w C, ponieważ:

- pomoże Ci to lepiej zrozumieć ogólną budowę i założenia Pythona;
- pewnego dnia możesz być zmuszony debugować lub utrzymywać rozszerzenie napisane w C lub C++;
- pomoże Ci to zrozumieć mechanizmy działania wysokopoziomowych narzędzi do budowania rozszerzeń.

Jak działają rozszerzenia w C i C++

Interpreter Pythona jest w stanie załadować rozszerzenia z dynamicznych/współdzielonych bibliotek, jeśli tylko udostępniają one odpowiedni interfejs z wykorzystaniem API Python/C. Definicja tego API jest załączana w kodzie źródłowym rozszerzenia za pomocą pliku nagłówkowego języka C o nazwie *Python.h* dystrybuowanego wraz z kodem źródłowym Pythona. W wielu dystrybucjach Linuksa plik nagłówkowy *Python.h* jest dostarczany w ramach osobnego pakietu oprogramowania (np. `python-dev` w systemach Debian/Ubuntu). W Windowsie plik nagłówkowy dostarczany jest domyślnie wraz z dystrybucją Pythona i znajduje się w katalogu *includes*/ pod ścieżką instalacyjną interpretera.

Interfejs Python/C tradycyjnie ulega drobnym zmianom z każdym wydaniem Pythona. W większości przypadków są to jedynie nowe funkcjonalności dodawane do API, przez co zachowywana jest wsteczna kompatybilność na poziomie źródeł. Niestety, wydania Pythona z reguły nie są binarnie wstecznie kompatybilne z powodu zmian w **binarnym interfejsie aplikacji** (ang. *Application Binary Interface* — ABI). Oznacza to, że rozszerzenie musi być zbudowane osobno dla każdej wspieranej wersji Pythona. Dodatkowo różne systemy operacyjne nie są między sobą binarnie kompatybilne, co czyni właściwie niemożliwym dostarczenie gotowej binarnej dystrybucji skompilowanego rozszerzenia działającej w każdym możliwym środowisku. Dlatego właśnie większość rozszerzeń dystrybuowana jest jedynie w postaci źródłowej.

Od Pythona w wersji 3.2 podzbiór interfejsu Python/C został określony jako posiadający stabilne ABI. Jest więc możliwe skompilowanie rozszerzenia opartego na ograniczonym API Python/C (tym o stabilnym ABI), które będzie działało ze wszystkimi wydaniami interpretera Pythona w wersjach 3.2 i wyższych. Ogranicza to jednak zakres dostępnych elementów API i nie rozwiązuje problemu dystrybucji rozszerzeń dla starszych wersji Pythona oraz dla różnych systemów operacyjnych. Jest to więc kompromis, przy którym traci się wyjątkowo dużo, a zyskuje wciąż niewiele.

Niezwykle ważną informacją, o której musisz wiedzieć, jest fakt, że API Python/C to funkcjonalność dostępna jedynie dla interpretera CPython. Twórcy głównych alternatywnych implementacji Pythona (PyPy, Jython oraz IronPython) od lat próbują umożliwić bezproblemowe wykorzystanie rozszerzeń, ale na chwilę obecną żadna z nich nie dostarcza w pełni działającego rozwiązania. Jedyną alternatywną implementacją Pythona oferującą pełne wsparcie dla rozszerzeń jest Stackless Python, ponieważ jest to tak naprawdę interpreter bazujący na zmodyfikowanych źródłach CPythona.

Rozszerzenia C/C++ dla Pythona muszą być skompilowane do postaci dynamicznych/współdzielonych bibliotek, zanim będą mogły być wykorzystane, ponieważ z oczywistych względów niemożliwe jest bezpośrednie zaimportowanie źródeł w C/C++ z poziomu kodu Pythona. Na szczęście biblioteki `distutils` i `setuptools` udostępniają odpowiednie klasy pomocnicze pozwalające zdefiniować rozszerzenia jako moduły pakietu Pythona. Dzięki temu rozszerzenia mogą być budowane i dystrybuowane za pomocą skryptu *setup.py*, tak jak zwykłe pakiety Pythona. Poniższy listing zawiera przykład skryptu *setup.py* z oficjalnej dokumentacji Pythona odpowiedzialny za pakowanie prostego pakietu zawierającego wbudowane rozszerzenia:

```
from distutils.core import setup, Extension


module1 = Extension(
    'demo',
    sources=['demo.c']
)


setup(
    name='PackageName',
    version='1.0',
```

```
    description='To jest pakiet demonstracyjny.',
    ext_modules=[module1]
)
```

Taki skrypt wymaga użycia dodatkowego polecenia podczas typowej procedury dystrybucyjnej:

```
python setup.py build
```

Polecenie to spowoduje skompilowanie wszystkich rozszerzeń podanych w liście argumentu `ext_modules` zgodnie z dodatkowymi ustawieniami kompilatora zawartymi w wywołaniu konstruktora klasy `Extension()`. Kompilatorem użytym w procesie budowy będzie domyślny kompilator dla Twojego środowiska. Sam krok kompilacji nie jest wymagany, jeśli rozszerzenie ma być dystrybuowane w formie źródłowej. W takim przypadku musisz się upewnić, że środowisko docelowe zawiera wszystkie składniki potrzebne w procesie kompilacji, takie jak kompilator, pliki nagłówkowe, oraz dodatkowe biblioteki potrzebne w czasie linkowania (jeśli Twoje rozszerzenie takowych wymaga). Więcej szczegółów na temat pakowania i dystrybucji rozszerzeń Pythona zawartych będzie w podrozdziale „Wyzwania związane z rozszerzeniami".

Dlaczego warto tworzyć rozszerzenia

Nie da się prosto odpowiedzieć na pytanie o to, kiedy warto zdecydować się na pisanie własnego rozszerzenia Pythona w języku C bądź C++. Praktyczna zasada mogłaby brzmieć: „Tylko wtedy, gdy nie ma innej opcji". Jest to jednak bardzo subiektywna reguła, gdyż pozostawia miejsce na osobistą interpretację tego, co jest, a co nie jest wykonywalne w Pythonie. W rzeczywistości trudno wskazać choćby jeden problem, którego nie dałoby się w całości wyeliminować, posługując się wyłącznie czystym kodem Pythona. Całkowicie inna kwestia to to, czy ostateczne rozwiązanie będzie wystarczająco dobre i wydajne. W praktyce rozszerzenia Pythona w C/C++ są szczególnie użyteczne w następujących sytuacjach:

- omijanie mechanizmu GIL (globalnej blokady interpretera) w programowaniu wielowątkowym;
- zwiększanie wydajności w krytycznych sekcjach kodu;
- integracja zewnętrznych bibliotek dynamicznych;
- integracja kodu napisanego w innych językach programowania;
- tworzenie własnych wbudowanych typów danych.

Nie oznacza to jednak, że którykolwiek z powyższych problemów jest nierozwiązywalny bez użycia rozszerzeń. Nawet tak kluczowe ograniczenie interpretera jak GIL może być przezwyciężone dzięki wykorzystaniu całkowicie innego podejścia do przetwarzania równoległego: zastosowania zielonych wątków bądź modelu przetwarzania opartego na wielu procesach.

Zwiększanie wydajności w krytycznych sekcjach kodu

Bądźmy szczerzy: głównym powodem, dla którego tak wielu programistów wybiera Pythona, na pewno nie jest wydajność. Kod Pythona nie wykonuje się szybko, za to możemy za jego pomocą niezwykle szybko tworzyć nowe oprogramowanie. Mimo wszystko, bez względu na to, jak wybitnymi programistami jesteśmy, prędzej czy później natrafimy na problem, którego wydajnie rozwiązać w Pythonie się nie da.

W większości przypadków rozwiązywanie problemów wydajności to kwestia doboru odpowiednich algorytmów i struktur danych, a nie zadanie ograniczania stałego narzutu wydajnościowego związanego z użytą technologią. Właściwie uciekanie się do rozszerzeń Pythona w celu ugrania kilku cyklów procesora w sytuacji, gdy kod jest napisany źle i nie wykorzystuje odpowiednich algorytmów, to niezwykle szkodliwa praktyka. Dosyć często wydajność kodu można znacznie poprawić bez potrzeby zwiększania złożoności projektu i dołączania nowego języka programistycznego do technologicznego stosu aplikacji. Jeśli to tylko możliwe, zwiększanie wydajności aplikacji powinno odbywać się przede wszystkim w ten sposób. Jednak w pewnych sytuacjach może się okazać, że kod napisany w Pythonie nawet z wykorzystaniem *najlepszych* znanych algorytmów i struktur danych nie będzie w stanie zmieścić się w pewnych arbitralnych limitach czasu wykonania bądź dostępnych zasobów.

Przykładową dziedziną biznesową nakładającą ściśle określone ograniczenia na wydajność aplikacji jest rynek handlu przestrzenią reklamową w modelu RTB (ang. *Real Time Bidding*). W skrócie model RTB opiera się na zakupie i sprzedaży internetowej przestrzeni reklamowej w modelu aukcyjnym przypominającym handel udziałami na giełdach. Handel przestrzenią reklamową odbywa się z reguły za pośrednictwem wyspecjalizowanej usługi giełdowej informującej w czasie rzeczywistym platformy popytowe (ang. *Demand Side Platform* — DSP) o pojawiających się możliwościach zakupu powierzchni. I to jest miejsce, w którym wszystko zaczyna być naprawdę ekscytujące. Większość giełd reklamowych na rynku RTB korzysta z protokołu OpenRTB (bazującego na HTTP) w komunikacji z potencjalnymi kupcami znajdującymi się po stronie DSP. W tym modelu to kupcy są faktycznymi serwerami HTTP, a giełda poprzez żądania informuje o nowych możliwościach kupna. Giełdy z reguły nakładają bardzo ścisłe ograniczenia dotyczące czasu, w jakim odpowiedź na żądanie musi być obsłużona (zwykle pomiędzy 50 a 100 ms). Ograniczenie to dotyczy całości komunikacji — od pierwszego bajta wysłanego przez giełdę aż po ostatni bajt odebrany w odpowiedzi, wliczając w to czas podróży pakietów TCP w obie strony. Aby jeszcze podgrzać atmosferę, warto wspomnieć, że dla platform DSP nie jest rzadkością obsługiwanie dziesiątek tysięcy żądań na sekundę. Możliwość przyspieszenia czasu przetwarzania żądań HTTP nawet o parę milisekund może decydować o sukcesie całego produktu w tej specyficznej branży. Oznacza to, że w określonych sytuacjach przeniesienie nawet prostego kodu do C może mieć praktyczny sens, oczywiście pod warunkiem, że przenoszony kod stanowi wąskie gardło wydajności i nie może już zostać ulepszony w żaden inny sposób. Zgodnie z tym, co kiedyś rzekł Guido van Rossum:

> „Jeśli potrzebujesz szybkości (…) — nie jesteś w stanie pobić pętli napisanej w C”.

Integracja kodu napisanego w innych językach programowania

W czasie krótkiej historii informatyki powstało wyjątkowo dużo użytecznych bibliotek. Byłoby wielką stratą zapominanie o całym tym dziedzictwie za każdym razem, gdy pojawia się nowy język programowania. Z drugiej strony, niemożliwe jest przenoszenie każdego istniejącego kawałka oprogramowania do wszystkich możliwych języków.

Języki C i C++ są najważniejszymi językami dostarczającymi ogrom bibliotek i wzorcowych implementacji wielu algorytmów, które mógłbyś chcieć zintegrować w swoich aplikacjach bez potrzeby przenoszenia ich kodu do Pythona. Na szczęście interpreter CPython jest napisany w C, dlatego najbardziej naturalna droga integracji takich bibliotek wiedzie przez własne rozszerzenia Pythona.

Integracja zewnętrznych bibliotek dynamicznych

Integracja kodu napisanego w innych językach programowania nie kończy się na C/C++. Wiele bibliotek, szczególnie oprogramowania osób trzecich o zamkniętym kodzie, dystrybuowanych jest w postaci skompilowanych plików binarnych. W języku C niezwykle łatwo jest ładować takie dynamiczne/współdzielone biblioteki i wywoływać ich funkcje. Oznacza to, że możesz korzystać z dowolnej skompilowanej biblioteki dynamicznej, jeśli tylko opakujesz ją odpowiednim rozszerzeniem za pomocą API Python/C.

Oczywiście nie jest to jedyna dostępna droga. Istnieją narzędzia (takie jak `ctypes` czy `cffi`), które pozwalają na ładowanie dynamicznych bibliotek za pomocą czystego kodu Pythona oraz bez konieczności pisania własnych rozszerzeń w C. W wielu przypadkach rozszerzenia opierające się na interfejsie Python/C wciąż pozostają dużo lepszym wyborem, ponieważ pozwalają na lepszą separację warstwy integracji (napisanej w C) od reszty Twojej aplikacji.

Tworzenie własnych wbudowanych typów danych

Python dostarcza bardzo wszechstronnej kolekcji wbudowanych typów. Niektóre z nich są oparte na najnowocześniejszych implementacjach (przynajmniej w CPythonie), które w dodatku są specjalnie dopasowane do wykorzystania w kontekście tego języka. Liczba dostępnych typów i kontenerów może być imponująca dla nowicjuszy, ale nie pokrywają one wszystkich możliwych potrzeb użytkowników.

Możesz oczywiście tworzyć wiele nowych struktur danych bezpośrednio w Pythonie, opierając je na istniejących już typach wbudowanych lub definiując od zera za pomocą własnych klas. Niestety, tak zdefiniowane struktury danych w specyficznych przypadkach mogą nie oferować wystarczającej wydajności. Cała siła złożonych struktur danych (takich jak `dict`) bierze się właśnie

z wewnętrznej implementacji interpretera. W przypadku CPythona korzystamy więc z wydajnej implementacji w języku C. Dlaczego by nie pójść tą samą drogą i nie zdefiniować własnych typów danych również w C?

Pisanie rozszerzeń

Zgodnie z tym, co zostało już powiedziane, pisanie rozszerzeń nie jest zadaniem prostym. Oferują one jednak w zamian wiele zalet. Najprostszą i rekomendowaną techniką tworzenia rozszerzeń jest wykorzystanie zaawansowanych narzędzi takich jak Cython czy Pyrex lub (w przypadku integracji kodu/bibliotek innych języków) oparcie swojego kodu na bibliotekach `ctypes` lub `cffi`. Projekty te znacznie zwiększą Twoją produktywność i sprawią, że kod rozszerzenia będzie łatwiejszy w rozwoju, czytaniu oraz utrzymaniu.

Mimo wszystko, jeśli jest to dla Ciebie nowy temat, dobrze, żebyś zaczął swoją przygodę z rozszerzeniami od napisania jakiegoś z wykorzystaniem jedynie języka C i interfejsu Python/C. Poprawi to Twoje zrozumienie działania rozszerzeń i sprawi, że docenisz zalety alternatywnych rozwiązań. Przez wzgląd na prostotę weźmiemy na warsztat pewien prosty problem algorytmiczny i spróbujemy zaimplementować jego rozwiązanie w postaci rozszerzenia Pythona na dwa różne sposoby:

- tworząc rozszerzenie bezpośrednio w języku C;
- korzystając z języka Cython.

Naszym zadaniem będzie napisanie funkcji wyznaczającej *n*-ty wyraz ciągu Fibonacciego. Potrzeba tworzenia rozszerzenia wyłącznie w celu implementacji takiej funkcji jest mało prawdopodobna. Jednakże jest to problem na tyle prosty, że będzie świetnie służył jako podstawa wiązania kodu w języku C z Pythonem za pomocą interfejsu Python/C. Naszym celem jest przede wszystkim uzyskanie przejrzystego i prostego rozwiązania, dlatego nie będziemy starali się wykorzystać metody optymalnej. Za wzorcową implementację posłuży nam następujący moduł Pythona:

```
"""Moduł Pythona dostarczający funkcję fibonacci."""


def fibonacci(n):
  """Zwróć n-ty wyraz ciągu Fibonacciego wyznaczony rekurencyjnie."""
    if n < 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Zauważ, że jest to jedna z najprostszych możliwych implementacji funkcji `fibonacci()` i z pewnością można by ją usprawnić na wiele sposobów. Nie będziemy jednak na tym etapie stosować żadnego z oczywistych ulepszeń (np. memoizacji), ponieważ nie są one celem naszego zadania. Na tej samej zasadzie będziemy się powstrzymywać od stosowania oczywistych optymalizacji na późniejszych etapach omawiania kodu w C oraz Cythonie, pomimo że te języki dostarczają ku temu więcej sposobności.

Zwyczajne rozszerzenia w C

Zanim w pełni zanurkujesz w tematykę rozszerzeń Pythona napisanych w C, należą Ci się słowa ostrzeżenia. Jeśli zamierzasz rozszerzać Pythona za pomocą C, powinieneś sprawnie posługiwać się oboma tymi językami programowania. Szczegółowa wiedza o mechanizmach języka jest ważna zwłaszcza w przypadku C. Brak biegłości w tym języku może prowadzić do prawdziwych katastrof ze względu na łatwość popełniania krytycznych błędów.

Jeśli zdecydowałeś się już na pisanie rozszerzeń Pythona w C, to zakładam, że znasz ten język w stopniu pozwalającym Ci zrozumieć proste przykłady zawarte w tym rozdziale. W kontekście C będziemy tłumaczyć jedynie zawiłości interfejsu Python/C oraz niektóre mechanizmy działania interpretera CPython. Wszystko dlatego, że jest to książka o Pythonie, a nie żadnym innym języku programowania. Jeśli nie potrafisz programować w C, zdecydowanie nie powinieneś próbować pisać w nim swojego pierwszego rozszerzenia. Lepiej zostawić to zadanie innym i skorzystać z bardziej przystępnych dla początkujących narzędzi Cython lub Pyrex. Interfejs Python/C, mimo że stworzony i zaprojektowany z wielką starannością, nie jest dobrym materiałem wprowadzającym do języka C.

Tak jak zostało to zapowiedziane wcześniej, spróbujemy przenieść naszą pythonową funkcję `fibonacci()` do języka C i udostępnić nasze rozwiązanie interpreterowi Pythona. Surowa implementacja funkcji w C (tj. bez kodu wiążącego ją z interpreterem za pomocą interfejsu Python/C) będzie analogiczna do wcześniejszego przykładu w Pythonie:

```
long long fibonacci(unsigned int n) {
    if (n < 2) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

Poniższy listing zawiera kod w pełni funkcjonalnego rozszerzenia udostępniającego powyższą funkcję w skompilowanym module:

```
#include <Python.h>


long long fibonacci(unsigned int n) {
    if (n < 2) {
        return 1;
    } else {
        return fibonacci(n-2) + fibonacci(n-1);
    }
}


static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
```

```
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
        result = Py_BuildValue("L", fibonacci((unsigned int)n));
    }

    return result;
}


static char fibonacci_docs[] =
    "fibonacci(n): Zwróć n-ty wyraz ciągu Fibonacciego "
    " wyznaczony rekurencyjnie.\n";


static PyMethodDef fibonacci_module_methods[] = {
    {"fibonacci", (PyCFunction)fibonacci_py,
     METH_VARARGS, fibonacci_docs},
    {NULL, NULL, 0, NULL}
};


static struct PyModuleDef fibonacci_module_definition = {
    PyModuleDef_HEAD_INIT,
    "fibonacci",
    "Moduł rozszerzenia dostarczający funkcję fibonacci.",
    -1,
    fibonacci_module_methods
};


PyMODINIT_FUNC PyInit_fibonacci(void) {
    Py_Initialize();

    return PyModule_Create(&fibonacci_module_definition);
}
```

Powyższy przykład może być nieco przytłaczający, ponieważ wyeksponowanie prostej w założeniu funkcji wymagało użycia co najmniej czterokrotności początkowego kodu. Nie martw się, zaraz omówię każdy z elementów tego kodu. Zanim jednak przystąpię do tej analizy, pokażę jeszcze, jak zamknąć powyższy kod w postaci pakietu gotowego do dystrybucji. Minimalna konfiguracja `setuptools` będzie wymagała użycia konstruktora klasy `setuptools.Extension` w celu poinformowania interpretera o tym, które pliki źródłowe rozszerzenia wymagają kompilacji:

```
from setuptools import setup, Extension


setup(
    name='fibonacci',
```

```
    ext_modules=[
        Extension('fibonacci', ['fibonacci.c']),
    ]
)
```

Proces kompilacji rozszerzenia może być jawnie zainicjowany za pomocą polecenia `build` skryptu *setup.py*, ale zostanie również wykonany automatycznie podczas instalacji pakietu. Poniższy listing przedstawia rezultat instalacji naszego rozszerzenia w trybie roboczym oraz zapis interaktywnej sesji interpretera z użyciem tak skompilowanego modułu:

```
$ ls -1a
fibonacci.c
setup.py

$ pip install -e .
Obtaining file:///Users/swistakm/dev/book/chapter7
Installing collected packages: fibonacci
  Running setup.py develop for fibonacci
Successfully installed Fibonacci

$ ls -1ap
build/
fibonacci.c
fibonacci.cpython-35m-darwin.so
fibonacci.egg-info/
setup.py

$ python
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import fibonacci
>>> help(fibonacci.fibonacci)

Help on built-in function fibonacci in fibonacci:

fibonacci.fibonacci = fibonacci(...)
    fibonacci(n): Zwróć n-ty wyraz ciągu Fibonacciego wyznaczony rekurencyjnie.

>>> [fibonacci.fibonacci(n) for n in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>>
```

Bliższe spojrzenie na interfejs Python/C

Wiesz już, jak poprawnie spakować, skompilować oraz zainstalować własne rozszerzenie Pythona w języku C. Wiesz również, że działa ono tak, jak oczekiwaliśmy. Nadszedł więc czas, aby przyjrzeć się dokładnie poszczególnym elementom jego kodu.

Moduł rozszerzenia rozpoczyna się od dyrektywy preprocesora C dołączającej plik nagłówkowy *Python.h*:

```
#include <Python.h>
```

Dyrektywa ta załącza definicję całego interfejsu Python/C i jest wszystkim, czego potrzebujesz, aby móc napisać własne rozszerzenie. W bardziej realistycznych przypadkach Twój kod będzie wymagał użycia większej liczby dyrektyw `include`, abyś mógł wykorzystać elementy standardowej biblioteki języka C oraz funkcje z pozostałych bibliotek. Nasz przykład jest wyjątkowo prosty, dlatego nie wymaga załączenia żadnych innych plików nagłówkowych.

Dalej w kodzie znajduje się kluczowy element naszego modułu:

```
long long fibonacci(unsigned int n) {
    if (n < 2) {
        return 1;
    } else {
        return fibonacci(n - 2) + fibonacci(n - 1);
    }
}
```

Powyższa funkcja `fibonacci()` jest jedyną użyteczną częścią naszego kodu wykonującą jakąkolwiek pracę. Jest to czysta implementacja C, której Python nie jest w stanie domyślnie zrozumieć. Reszta kodu będzie odpowiadała za stworzenie odpowiedniej warstwy interfejsu eksponującej powyższą funkcję za pomocą API Python/C.

Pierwszym krokiem w udostępnieniu naszego kodu języka C interpreterowi CPythona jest stworzenie funkcji języka C kompatybilnej z interfejsem tego interpretera. W Pythonie wszystko jest obiektem. Oznacza to, że funkcja C wywoływana w Pythonie musi również zwracać prawdziwe obiekty Pythona. Interfejs Python/C dostarcza typ `PyObject` i każda funkcja wywoływana przez interpreter musi zwracać wskaźnik zmiennej tego typu. Sygnatura naszej funkcji jest następująca:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args)
```

Zwróć uwagę na fakt, że powyższa sygnatura nie określa dokładnej listy przyjmowanych argumentów, a jedynie obiekt `PyObject* args` będący wskaźnikiem krotki z przekazanymi do funkcji wartościami. Ostateczna weryfikacja przekazanej listy argumentów musi się odbyć w ciele samej funkcji i tym właśnie zajmuje się funkcja `fibonacci_py()`. Analizuje listę argumentów `args`, zakładając, że zawiera ona tylko jeden element typu `unsigned int`, i przekazując jego wartość do wywołania `fibonacci()` w celu uzyskania wartości wskazanego wyrazu ciągu Fibonacciego:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
```

```
            result = Py_BuildValue("L", fibonacci((unsigned int)n));
        }

        return result;
    }
```

> Powyższy przykład zawiera pewne poważne błędy, które z łatwością powinien dostrzec doświadczony programista. Spróbuj je odnaleźć na własną rękę w ramach ćwiczenia pracy z rozszerzeniami w C. Dla zwięzłości pozostawię chwilowo kwestię wspomnianych błędów nierozwiązaną. Zajmę się nimi później, podczas omawiania obsługi błędów oraz wyjątków.

Literał znakowy `"l"` w wywołaniu `PyArg_ParseTuple(args, "l", &n)` oznacza, że lista argumentów `args` powinna zawierać jedynie pojedynczą wartość typu `long`. W przypadku błędu wywołanie to zwróci wartość `NULL` i zapisze informacje o wyjątku w odpowiednim globalnym miejscu przechowywania informacji o błędach, niezależnym dla każdego wątku interpretera. Szczegółami mechanizmu obsługi wyjątków zajmę się w podrozdziale „Obsługa wyjątków".

Właściwa sygnatura funkcji odpowiedzialnej za analizę argumentów to `int PyArg_ParseTuple ↪(PyObject *args, const char *format, ...)`. Zaraz za argumentem `format` przyjmuje ona listę wskaźników dla parametrów wyjściowych. Lista ta może mieć zmienną długość. Mechanizm działania jest analogiczny do funkcji `scanf()` dostępnej w standardowej bibliotece języka C. Jeśli nasze założenie okaże się nieprawidłowe i użytkownik przekaże niewłaściwą listę argumentów, to `PyArg_ParseTuple()` spowoduje podniesienie odpowiedniego wyjątku. Jest to bardzo komfortowy sposób opisu sygnatur funkcji, trzeba się tylko do niego przyzwyczaić. Oczywiście nie jest to sposób tak wygodny jak ten stosowany bezpośrednio w kodzie Pythona. W dodatku tak zdefiniowane sygnatury nie mogą być analizowane za pomocą typowych narzędzi introspekcji obiektów języka dostępnych w Pythonie. Musisz pamiętać o tej wadzie, decydując się na tworzenie rozszerzeń w języku C.

Wspominałem już o tym, że interpreter Pythona wymaga, aby funkcje rozszerzeń zwracały obiekty. Oznacza to, że rezultatem wywołania funkcji `fibonacci_py()` nie może być surowa wartość typu `long long` otrzymana z funkcji `fibonacci()`. Próba zapisania kodu w ten sposób spowodowałaby błąd kompilacji, ponieważ niemożliwe jest automatyczne rzutowanie typów pomiędzy obiektami Pythona a podstawowymi typami języka C. Zamiast tego należy zastosować funkcję `Py_BuildValue(*format, ...)`, która jest odpowiednikiem funkcji `PyArg_ParseTuple()` i przyjmuje podobny zbiór argumentów. Główna różnica jest taka, że parametry zawarte po argumencie `format` stanowią parametry wejściowe, a nie wyjściowe. W związku z tym należy użyć wartości zmiennych, a nie ich wskaźników.

Po zdefiniowaniu funkcji `fibonacci_py()` większość pracy mamy już za sobą. Ostatnim krokiem jest przeprowadzenie odpowiedniej inicjalizacji modułu oraz dostarczenie odpowiednich metadanych sprawiających, że korzystanie z modułu będzie łatwiejsze dla użytkowników. Jest to stały element kodu każdego rozszerzenia i w przypadku prostych modułów może być objętościowo większy niż właściwy kod, który chcemy udostępnić w Pythonie. Zwykle składa się z paru statycznych struktur danych oraz jednej funkcji inicjalizacyjnej wywoływanej podczas importu modułu.

Na początku tworzymy statyczny ciąg znaków zawierający docstring funkcji `fibonacci_py()`:

```
static char fibonacci_docs[] =
    "fibonacci(n): Zwróć n-ty wyraz ciągu Fibonacciego "
    " wyznaczony rekurencyjnie.\n";
```

Zwróć uwagę na to, że powyższy ciąg znaków mógłby być później *osadzony* bezpośrednio w zmiennej `fibonacci_module_methods`. Dobrą praktyką jest jednak odseparowanie docstringów od siebie i przechowywanie ich blisko kodu funkcji, które opisują.

Następną częścią kodu naszego rozszerzenia jest tablica struktur typu `PyMethodDef` definiujących metody (funkcje) dostępne w module. Struktura `PyMethodDef` zawiera dokładnie cztery pola:

- `char* ml_name` — definiuje nazwę funkcji.
- `PyCFunction ml_meth` — jest wskaźnikiem funkcji C.
- `int ml_flags` — jest polem bitowym zawierającym flagi określające konwencję wywoływania lub wiązania funkcji. Konwencje wiązania mają zastosowanie tylko do metod klas.
- `char* ml_doc` — jest wskaźnikiem ciągu znaków dokumentującego funkcję/metodę.

Tablica ta zawsze musi się kończyć wartością strażniczą `{NULL, NULL, 0, NULL}` wskazującą na koniec struktury. W naszym przypadku utworzyliśmy statyczną tablicę `PyMethodDef fibonacci_module_methods[]` składającą się tylko z dwóch elementów (wliczając wartość strażniczą):

```
static PyMethodDef fibonacci_module_methods[] = {
    {"fibonacci", (PyCFunction)fibonacci_py,
     METH_VARARGS, fibonacci_docs},
    {NULL, NULL, 0, NULL}
};
```

Poszczególne elementy pierwszego wpisu przekładają się na pola struktury `PyMethodDef` w następujący sposób:

- `ml_name = "fibonacci"` — funkcja C `fibonacci_py()` będzie dostępna w module Pythona pod nazwą `fibonacci`.
- `ml_meth = (PyCFunction)fibonacci_py` — rzutowanie na wskaźnik typu `PyCFunction` jest wymagane przez strukturę interfejsu Python/C i wynika z konwencji wywoływania zdefiniowanej dalej za pomocą pola `ml_flags`.
- `ml_flags = METH_VARARGS` — `METH_VARARGS` definiuje konwencję wywoływania pozwalającą na wywołanie funkcji ze zmienną listą argumentów pozycyjnych bez możliwości stosowania argumentów kluczowych.
- `ml_doc = fibonacci_docs` — funkcja modułu Pythona zostanie udokumentowana za pomocą ciągu znakowego `fibonacci_docs`.

Po skompletowaniu tablicy funkcji możemy przystąpić do tworzenia struktury danych definiującej cały moduł Pythona. Do tego celu należy wykorzystać typ danych `PyModuleDef` zawierający wiele pól. Część z nich jest stosowana jedynie w złożonych przypadkach, gdy konieczna jest

bardzo dokładna kontrola nad procesem inicjalizacji modułu. W naszym przykładzie będziemy zainteresowani tylko pięcioma polami tej struktury:

- `PyModuleDef_Base m_base` — zawsze powinno być zainicjalizowane wartością `PyModuleDef_HEAD_INIT`.
- `char* m_name` — określa nazwę modułu udostępnianego w ramach rozszerzenia.
- `char* m_doc` — jest to wskaźnik ciągu znaków dokumentującego zawartość modułu. W większości przypadków tylko jeden moduł Pythona jest implementowany w pojedynczym pliku źródłowym rozszerzenia, dlatego zwykle zawartość tego ciągu osadza się bezpośrednio w strukturze.
- `Py_ssize_t m_size` — jest to rozmiar przestrzeni pamięci zaalokowanej na potrzeby przechowywania dodatkowego stanu modułu. Przestrzeń ta potrzebna będzie jedynie, gdy konieczna jest szczególna obsługa wielu interpreterów lub wykorzystywana jest wielofazowa inicjalizacja modułu. W większości przypadków nie będziesz korzystać z tej funkcjonalności i pole to przyjmie wartość `-1`.
- `PyMethodDef* m_methods` — jest to wskaźnik tablicy przechowującej opisy funkcji modułu Pythona utworzone z wykorzystaniem typu `PyMethodDef`. Jeśli moduł nie zawiera żadnych funkcji, pole to przyjmuje wartość `NULL`. W naszym przypadku będzie to tablica `fibonacci_module_methods`.

Opis pozostałych pól niewymaganych w naszym prostym przykładzie znajdziesz w oficjalnej dokumentacji Pythona (*https://docs.python.org/3/c-api/module.html*). Niewymagane pola należy jawnie zainicjalizować wartością `NULL` lub całkowicie pominąć. Opcjonalność pozostałych pól sprawia, że możemy posłużyć się uproszczoną pięcioelementową definicją modułu `fibonacci`:

```
static struct PyModuleDef fibonacci_module_definition = {
    PyModuleDef_HEAD_INIT,
    "fibonacci",
    "Moduł rozszerzenia zawierający implementację ciągu Fibonacciego. ",
    -1,
    fibonacci_module_methods
};
```

Ostatnim elementem wieńczącym naszą pracę jest funkcja inicjalizacyjna modułu. Musi ona wykorzystywać ściśle określoną konwencję nazewniczą. To dzięki niej interpreter Pythona będzie w stanie ją znaleźć po załadowaniu dynamicznej/współdzielonej biblioteki. Nazwa funkcji inicjalizacyjnej powinna mieć formę `PyInit_<nazwa_modułu>`. Ciąg `<nazwa>` powinien być dokładnie tym samym ciągiem znaków co ciąg użyty jako wartość pola `m_base` struktury `PyModuleDef` oraz pierwszy argument konstruktora klasy `setuptools.Extension`. Jeśli Twój moduł nie wymaga skomplikowanej inicjalizacji, funkcja inicjalizacyjna przyjmuje prostą i krótką formę:

```
PyMODINIT_FUNC PyInit_fibonacci(void) {
    return PyModule_Create(&fibonacci_module_definition);
}
```

PyMODINIT_FUNC jest makrem preprocesora, które ustala PyObject* jako typ wyjściowy funkcji inicjalizacyjnej oraz zapewnia odpowiednie deklaracje dla linkera, jeśli takowe są wymagane przez docelową platformę systemową.

Konwencje wywoływania i wiązania funkcji

W podrozdziale „Bliższe spojrzenie na interfejs Python/C" wspomniałem o polu bitowym ml_flags struktury PyMethodDef służącym do definiowania konwencji wywoływania i wiązania funkcji. **Flagi służące do określania konwencji wywoływania** to:

- METH_VARARGS — jest to typowa konwencja wywoływania dla funkcji Pythona akceptujących jedynie pozycyjne argumenty podczas wywoływania. Wartości pola ml_meth dla funkcji korzystających z tej konwencji powinny przyjmować typ PyCFunction. Funkcje C tej konwencji otrzymują dwa argumenty typu PyObject*: obiekt self (jeśli funkcja jest metodą) lub obiekt modułu (jeśli funkcja jest funkcją z przestrzeni nazw modułu). Typowa sygnatura dla funkcji tej konwencji to PyObject* function(PyObject* self, PyObject* args).
- METH_KEYWORDS — jest to konwencja wywoływania dla funkcji przyjmujących argumenty kluczowe podczas wywoływania. Powiązanym typem dla pola ml_meth jest PyCFunctionWithKeywords. Funkcje C tej konwencji otrzymują trzy argumenty typu PyObject*: obiekt self, krotkę args oraz słownik argumentów kluczowych. W połączeniu z flagą METH_VARARGS pierwsze dwa argumenty mają taką samą semantykę co argumenty funkcji konwencji METH_VARARGS. W przeciwnym razie argument args będzie przyjmował wartość NULL. Typową sygnaturą dla funkcji tej konwencji jest PyObject* function(PyObject* self, PyObject* args, PyObject* keywds)
- METH_NOARGS — jest to konwencja wywoływania dla funkcji, które nie przyjmują żadnych argumentów. Powiązanym typem dla pola ml_meth jest PyCFunction, a więc sygnatura funkcji pozostaje taka sama jak w przypadku konwencji METH_VARARGS (tj. dwa argumenty: self oraz args). Jedyna różnica polega na tym, że argument args zawsze przyjmuje wartość NULL, więc nie ma potrzeby wywoływania PyArg_ParseTuple() w ciele funkcji tej konwencji. Flaga METH_NOARGS nie może być łączona z żadnymi innymi flagami konwencji wywoływania.
- METH_O — jest to skrócona konwencja dla funkcji i metod przyjmujących wyłącznie jeden argument. Powiązanym typem dla pola ml_meth jest PyCFunction, a więc sygnatura funkcji pozostaje taka sama jak w przypadku konwencji METH_VARARGS (tj. dwa argumenty: self oraz args). Główna różnica polega na tym, że wartość przekazana w argumencie args będzie pojedynczym obiektem Pythona, a nie krotką argumentów. Dlatego nie należy korzystać z funkcji PyArg_ParseTuple() w ciałach funkcji tej konwencji. Flaga METH_O również nie może być łączona z żadnymi innymi flagami konwencji wywoływania.

Funkcje przyjmujące argumenty kluczowe są opisywane za pomocą flagi METH_KEYWORDS lub bitowej kombinacji dwóch flag w postaci METH_VARARGS | METH_KEYWORDS. Funkcje tej konwencji powinny rozpakowywać swoje argumenty z wykorzystaniem PyArg_ParseTupleAndKeywords()

zamiast funkcji PyArg_ParseTuple() lub PyArg_UnpackTuple(). Poniższy listing zawiera przykład prostego modułu rozszerzenia z jedną funkcją zwracającą wartość None. Funkcja ta przyjmuje dwa nazwane argumenty kluczowe i wypisuje je do standardowego wyjścia:

```
#include <Python.h>

static PyObject* print_args(PyObject *self, PyObject *args, PyObject *keywds)
{
    char *first;
    char *second;

    static char *kwlist[] = {"first", "second", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "ss", kwlist,
                                     &first, &second))
        return NULL;

    printf("%s %s\n", first, second);

    Py_INCREF(Py_None);
    return Py_None;
}


static PyMethodDef module_methods[] = {
    {"print_args", (PyCFunction)print_args,
     METH_VARARGS | METH_KEYWORDS,
     "Wypisz przekazane argumenty."},
    {NULL, NULL, 0, NULL}
};


static struct PyModuleDef module_definition = {
    PyModuleDef_HEAD_INIT,
    "kwargs",
    "Przykład przetwarzania argumentów kluczowych.",
    -1,
    module_methods
};


PyMODINIT_FUNC PyInit_kwargs(void) {
    return PyModule_Create(&module_definition);
}
```

Mechanizm analizowania argumentów funkcji Pythona w rozszerzeniach korzystających z interfejsu Python/C jest bardzo elastyczny, a jego szczegółowa oficjalna dokumentacja dostępna jest pod adresem: *https://docs.python.org/3.5/c-api/arg.html*. Parametr format funkcji PyArg_ParseTuple(PyObject *args, const char *format, ...) oraz PyArg_ParseTupleAndKeywords

`↪(PyObject *args, PyObject *kw, const char *format, char *keywords[], ...)` pozwala na drobnoziarnistą kontrolę nad liczbą i typami przekazywanych argumentów. Każda dopuszczalna w Pythonie konwencja przekazywania argumentów jest możliwa do zakodowania w rozszerzeniach C, włączając w to przede wszystkim:

- funkcje z domyślnymi wartościami argumentów;
- funkcje z argumentami określonymi jako wyłącznie kluczowe;
- funkcje przyjmujące zmienną liczbę argumentów.

Flagi określające konwencje wiązania, czyli `METH_CLASS`, `METH_STATIC` oraz `METH_COEXIST`, zarezerwowane są wyłącznie dla metod i nie mogą być użyte do opisania funkcji z przestrzeni nazw modułu. Dwie pierwsze powinny być dosyć oczywiste — stanowią odpowiedniki dekoratorów `classmethod` oraz `staticmethod` i zmieniają znaczenie argumentu `self` przekazanego do funkcji języka C.

Konwencja `METH_COEXIST` pozwala na ładowanie metod w miejsce istniejących definicji. Jest wykorzystywana bardzo rzadko. Znajduje zastosowanie w sytuacjach, gdy chcesz dostarczyć własną implementację metody w C, która normalnie zostałaby wygenerowana automatycznie na podstawie innych dostępnych w obiekcie metod. Oficjalna dokumentacja Pythona wymienia przykład metody `__contains__()`, która będzie wygenerowana automatycznie, jeśli definicja typu zawiera zdefiniowany slot `sq_contains`. Definiowanie własnych typów z użyciem interfejsu Python/C leży daleko poza zakresem materiału zawartego w tej książce.

Obsługa wyjątków

Język C, w odróżnieniu od Pythona i C++, nie oferuje odpowiedniej składni pozwalającej na podnoszenie i przechwytywanie wyjątków. Cała obsługa błędów w języku C opiera się na zwracaniu specjalnych wartości wyjściowych oraz na ewentualnych globalnych zmiennych przechowujących szczegóły ostatniego błędu.

Obsługa błędów w rozszerzeniach opartych na interfejsie Python/C zbudowana jest właśnie na powyższych prostych zasadach. Każdy wątek interpretera posiada własną globalną strukturę przechowującą informacje o błędach, które wystąpiły podczas wywoływania funkcji API Python/C. Wspomniana struktura przyjmuje wartość będącą opisem błędu. O fakcie wystąpienia błędu dowiadujesz się dzięki weryfikacji wartości zwracanych przez wywołane funkcje. Interfejs Python/C korzysta z ustandaryzowanego mechanizmu informacji o błędach:

- Jeśli funkcja zwraca wskaźnik, błąd jest wskazywany za pomocą wartości `NULL`.
- Jeśli funkcja zwraca wartości typu `int`, błąd jest wskazywany za pomocą wartości `-1`.

Jedynym wyjątkiem od powyższych zasad w interfejsie Python/C są funkcje `PyArg_*()`, które zwracają wartość `1`, aby poinformować o sukcesie, oraz wartość `0`, aby poinformować o błędzie.

Aby przyjrzeć się powyższemu mechanizmowi w praktyce, przypomnijmy sobie funkcję `fibo↪nacci_py()` z poprzednich podrozdziałów:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;

    if (PyArg_ParseTuple(args, "l", &n)) {
      result = Py_BuildValue("L", fibonacci((unsigned int) n));
    }

    return result;
}
```

Linie kodu biorące udział w obsłudze błędów zostały dodatkowo wyróżnione pogrubieniem. Obsługa błędów rozpoczyna się już od linii, w której zmienna `result` przechowująca zwracany wynik funkcji jest zainicjalizowana wartością `NULL`. Wiemy już, że wartość `NULL` informuje o błędzie. Zakładanie błędu jako domyślnego stanu aplikacji jest jedną z charakterystycznych cech kodu rozszerzeń Pythona.

Dalej mamy wywołanie funkcji `PyArg_ParseTuple()`, która w przypadku wystąpienia wyjątku ustawi informację o błędzie w globalnej strukturze interpretera i zwróci wartość `0`. Sprawdzanie błędu jest częścią głównej instrukcji `if`, dlatego jeśli wystąpi błąd, funkcja nie zrobi nic więcej i zwróci wartość `NULL`. Ktokolwiek wywołał funkcję `fibonacci_py()`, zostanie poinformowany o wystąpieniu błędu.

Funkcja `Py_BuildValue()` również może podnieść wyjątek. Zwraca ona obiekty typu `PyObject*` (wskaźnik), dlatego w przypadku awarii zwróci wartość `NULL`. Możemy przechować tę wartość w zmiennej `result` i zwrócić ją dalej.

Nasza praca nie kończy się oczywiście na troszczeniu się o wyjątki podnoszone przez wywołania funkcji interfejsu Python/C. Jest bardzo prawdopodobne, że będziesz też chciał informować użytkownika modułu rozszerzenia o innych błędach, które mogą się pojawić podczas jego użytkowania. Interfejs Python/C udostępnia wiele funkcji pozwalających na podnoszenie własnych wyjątków, a najprostszą z nich i najczęściej używaną jest `PyErr_SetString()`. Zapisuje ona wskaźnik wystąpienia błędu, posługując się wskazanym typem wyjątku oraz przekazanym ciągiem znaków stanowiącym szczegółowy opis błędu. Pełna sygnatura tej funkcji jest następująca:

```
void PyErr_SetString(PyObject* type, const char* message)
```

Na początku podrozdziału „Zwyczajne rozszerzenia w C" przyznałem, że kod naszej przykładowej funkcji `fibonacci_py()` zawiera pewne poważne błędy. Teraz nadszedł czas, aby się nimi zająć. Na szczęście dysponujemy już odpowiednimi do tego zadania narzędziami. Błąd polega na niebezpiecznym rzutowaniu wartości typu `long` na zmienną typu `unsigned int` w poniższych liniach kodu:

```
if (PyArg_ParseTuple(args, "l", &n)) {
    result = Py_BuildValue("L", fibonacci((unsigned int) n));
}
```

Wywołanie funkcji `PyArg_ParseTuple()` zapewnia, że pierwszy i jedyny argument funkcji zostanie zinterpretowany jako wartość typu `long` (specyfikator typu `"l"`). Wartość ta jest później jawnie rzutowana na typ `unsigned int` (liczba całkowita bez znaku). Problem wystąpi, gdy użytkownik rozszerzenia z poziomu Pythona wywoła funkcję `fibonacci()` z ujemnym argumentem. Przykładowo: 32-bitowa wartość całkowita `-1` zostanie zinterpretowana jako `4294967295` po rzutowaniu na 32-bitowy typ całkowity bez znaku. Taka wartość spowoduje wystąpienie zbyt głębokiego wywołania rekurencyjnego, co z kolei wiąże się z przepełnieniem stosu i błędem naruszenia pamięci. Zwróć uwagę na to, że tego typu problem wystąpi również w przypadku podania zbyt wysokiego argumentu funkcji `fibonacci()`. Nie usuniemy tego problemu bez całkowitego przeprojektowania naszej funkcji — dodania rozszerzeń. Możemy za to niskim kosztem zapewnić, że argument wejściowy będzie spełniał pewne narzucone z góry ograniczenia. Załóżmy, że oczekujemy argumentów o wartości większej bądź równej `0`, a w przypadku niespełnienia tego ograniczenia podnosimy wyjątek typu `ValueError`:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;
    long long fib;

    if (PyArg_ParseTuple(args, "l", &n)) {
        if (n<0) {
            PyErr_SetString(PyExc_ValueError,
                            "Wartość n nie może być mniejsza niż 0");
        } else {
            result = Py_BuildValue("L", fibonacci((unsigned int)n));
        }
    }

    return result;
}
```

Pamiętaj na koniec, że globalna struktura przechowująca informacje o błędzie nie wyczyści swojego stanu sama. Część błędów będziesz mógł bezproblemowo obsłużyć bezpośrednio w swoim kodzie C (w sposób analogiczny do konstrukcji `try ... except` w kodzie Pythona). W takich przypadkach możesz chcieć wyczyścić globalny stan błędów, który nie będzie już aktualny. Służy do tego funkcja `PyErr_Clear()`.

Zwalnianie blokady GIL

Wspomniałem już wcześniej, że rozszerzenia w C mogą być sposobem, aby ominąć mechanizm GIL. Jest to sławne ograniczenie interpretera CPython, sprawiające, że tylko jeden wątek interpretera może w określonym czasie wykonywać kod Pythona. Przetwarzanie równoległe w modelu opartym na wielu procesach jest najczęściej rekomendowanym rozwiązaniem tego problemu, jednak takie podejście może nie być najlepsze dla niektórych wyjątkowo równoległych algorytmów ze względu na wysoki narzut zasobów konieczny przy uruchamianiu i komunikowaniu wielu niezależnych procesów interpretera.

Ponieważ rozszerzenia wykorzystywane są najczęściej w sytuacjach, gdzie większość pracy wykonuje się w czystym C bez wielu wywołań funkcji interfejsu Python/C, możliwe jest (a nawet zalecane) jawne zwalnianie blokady GIL w wybranych sekcjach aplikacji. Dzięki temu możesz korzystać z zalet posiadania wielu procesorów lub rdzeni procesora w aplikacjach wielowątkowych. Zwalnianie blokady GIL jest wyjątkowo proste i wymaga owinięcia odpowiednich sekcji kodu za pomocą właściwych makr preprocesora udostępnionych przez interfejs Python/C. Niezwykle ważne jest, aby sekcja kodu, w której blokada jest zwolniona, nie wykorzystywała żadnych struktur Pythona ani nie wywoływała funkcji API Python/C. Dwa makra służące do kontroli nad mechanizmem GIL to:

- `Py_BEGIN_ALLOW_THREADS` — deklaruje ukrytą lokalną zmienną zapisującą aktualny stan wątku oraz zwalnia blokadę interpretera.
- `Py_END_ALLOW_THREADS` — nakłada z powrotem blokadę interpretera oraz przywraca stan wątku zapisany w ukrytej zmiennej lokalnej.

Jeśli przyjrzymy się dokładniej źródłom C naszego rozszerzenia `fibonacci`, zauważymy, że funkcja `fibonacci()` nie uruchamia bezpośrednio kodu Pythona i nie posługuje się jego strukturami. Z kolei funkcja `fibonacci_py()` owija jedynie wywołanie `fibonacci(n)` odpowiednim interfejsem Pythona. Możemy zwolnić globalną blokadę interpretera wokół tego wywołania w następujący sposób:

```
static PyObject* fibonacci_py(PyObject* self, PyObject* args) {
    PyObject *result = NULL;
    long n;
    long long fib;

    if (PyArg_ParseTuple(args, "l", &n)) {
        if (n<0) {
            PyErr_SetString(PyExc_ValueError,
                            "Wartość n nie może być mniejsza niż 0");
        } else {
            Py_BEGIN_ALLOW_THREADS;
            fib = fibonacci(n);
            Py_END_ALLOW_THREADS;

            result = Py_BuildValue("L", fib);
        }
    }

    return result;
}
```

Zliczanie referencji

W końcu przyszedł czas na podjęcie tematu zarządzania pamięcią w Pythonie. CPython posiada własny odśmiecacz (ang. *garbage collector*), zajmuje się on jednak wyłącznie wykrywaniem cyklicznych referencji i ma na celu wspieranie **zliczania referencji** jako podstawowego mechanizmu zarządzania pamięcią. Oba mechanizmy mają za zadanie zwalnianie pamięci, w której zapisane są niewykorzystywane już obiekty.

Dokumentacja interfejsu Python/C wprowadza pojęcie **prawa własności do obiektów** w celu wyjaśnienia, w jaki sposób następuje dealokacja obiektów. Obiekty w Pythonie nigdy nie są posiadane na wyłączność i zawsze pozostają współdzielone. Właściwe tworzenie obiektów (alokacja pamięci) jest zadaniem zarządcy pamięci. To jedyny komponent Pythona odpowiedzialny za alokację i dealokację pamięci dla obiektów przechowywanych na prywatnej stercie procesu interpretera. Tym, co faktycznie może być posiadane, są referencje do obiektów.

Każdy obiekt Pythona ma dodatkowy licznik referencji (wskaźników `PyObject*`). Kiedy licznik ten osiąga wartość 0, oznacza to, że nie istnieje już żadna poprawna referencja do niego i może zostać wywołany przypisany mu uchwyt dealokacji. Interfejs Python/C udostępnia dwa makra preprocesora służące do zwiększania i zmniejszania licznika referencji obiektów: `Py_INCREF()` oraz `Py_DECREF()`. Zanim się nimi zajmiemy, musimy poznać parę dodatkowych terminów związanych z prawem własności do referencji:

- **Przekazywanie prawa własności** — za każdym razem, gdy mówimy o przekazaniu prawa własności do referencji, oznacza to, że licznik referencji został zwiększony i obowiązkiem wywołującego funkcję jest zmniejszenie licznika, gdy obiekt nie jest już potrzebny. Większość funkcji interfejsu Python/C zwracających nowo utworzone obiekty (np. `Py_BuildValue`) przekazuje prawo własności do referencji. Jeśli obiekt ma być zwrócony dalej jako wskaźnik `PyObject*`, to prawo do własności będzie przekazane ponownie. W takim przypadku nie należy ręcznie zmniejszać licznika referencji, ponieważ nie jest to już obowiązkiem funkcji oddającej swoje prawo własności do referencji. Dlatego właśnie kod funkcji `fibobacci_py()` nie zawiera wywołania makra `Py_DECREF(result)`.
- **Pożyczone referencje** — *pożyczanie* referencji zachodzi najczęściej, gdy funkcja otrzymuje referencję do obiektu jako argument wywołania. Licznik referencji nie powinien się zwiększyć po wykonaniu funkcji. Może zwiększyć się chwilowo w trakcie jej wykonywania na potrzeby przetwarzania, ale ostatecznie powinien wrócić do początkowej wartości. W naszym przykładzie funkcji `fibonacci_py()` argumenty `self` i `args` stanowią właśnie pożyczone referencje. Niektóre funkcje API Python/C mogą także zwrócić pożyczone referencje. Funkcjami tymi są przede wszystkim `PyTuple_GetItem()` i `PyList_GetItem()`. Inne określenie tego typu wskaźników to **referencje niechronione**. Nie ma potrzeby zmieniania licznika referencji takich obiektów, chyba że mają one być zwrócone dalej w wyniku wywołania funkcji. Szczególnej uwagi wymagają pożyczone/niechronione referencje przekazywane jako parametry wywołania innych funkcji. W większości przypadków warto zabezpieczyć je dodatkową parą `Py_INCREF()`/`Py_DECREF()` wokół wywołania funkcji otrzymującej pożyczoną referencję.
- **Ukradzione referencje** — niektóre funkcje interfejsu Python/C zamiast pożyczać kradną przekazane przez argument referencje. Dzieje się tak w przypadku dwóch funkcji: `PyTuple_SetItem()` i `PyList_SetItem()`. Funkcje te w pełni przejmują prawo własności do przekazanych referencji. Nie zwiększają licznika przekazanych referencji, ale zmniejszą go, jeśli nie będą już potrzebować dostępu do obiektu.

Śledzenie liczników referencji jest jednym z najtrudniejszych aspektów pisania skomplikowanych rozszerzeń. Część związanych z nimi problemów może przez długi czas pozostać niezauważona — najczęściej, dopóki kod nie zostanie uruchomiony w wielowątkowej aplikacji.

Jeden z częstych problemów spowodowany jest naturą modelu obiektowości w Pythonie oraz faktem, że wiele funkcji interfejsu Python/C zwraca pożyczone referencje. Gdy licznik referencji osiąga wartość zerową, uruchamiany jest odpowiedni uchwyt dealokacji. Klasy zdefiniowane przez użytkownika mogą posiadać własną metodę `__del__()` wywoływaną dokładnie w tym momencie. Może to być dowolny kod Pythona i możliwe, że wpłynie on na inne obiekty oraz ich liczniki referencji. Oficjalna dokumentacja Pythona zawiera poniższy przykład kodu, który może być ofiarą opisanego zjawiska:

```
void bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

Powyższy kod wygląda na całkowicie nieszkodliwy. Problem polega na tym, że w momencie wywoływania funkcji `PyList_SetItem()` nie wiemy, jakie elementy są przechowywane wewnątrz listy `list`. Gdy funkcja `PyList_SetItem()` zapisuje nową wartość pod indeksem `list[1]`, obiekt listy musi pozbyć się prawa własności do referencji obiektu znajdującego się wcześniej pod tym samym indeksem. Jeśli była to jedyna istniejąca referencja do obiektu, to jego licznik referencji osiągnie wartość zerową i obiekt zostanie zdealokowany. W przypadku klas zdefiniowanych przez użytkownika możliwe jest uruchomienie w tym momencie dowolnego kodu Pythona w ramach metody `__del__()`. Poważny błąd może wystąpić, jeśli w ramach takiej metody zostanie usunięty z listy również obiekt `list[0]`, a jego licznik referencji także się wyzeruje. Zwróć uwagę na to, że `PyList_GetItem()` zwraca *pożyczone* referencje! Jest więc możliwe, że funkcja `PyObject_Print()` zostanie wywołana z referencją do już nieistniejącego obiektu. Spowoduje to błąd naruszenia pamięci i awaryjne zakończenie pracy interpretera.

Odpowiednim podejściem jest ręczne chronienie pożyczonych referencji przez cały czas, kiedy możliwa jest dealokacja dowolnego (nawet niepowiązanego) obiektu:

```
void no_bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

Cython

Cython jest optymalizującym statycznym kompilatorem Pythona, a także osobnym językiem programowania będącym nadzbiorem Pythona. Jako kompilator może przeprowadzać kompilację zarówno natywnego kodu Pythona, jak i własnego dialektu do formy rozszerzeń C opartych na interfejsie Python/C. Cython pozwala na połączenie mocy Pythona i C bez konieczności ręcznej obsługi interfejsu Python/C.

Cython jako kompilator kodu Pythona

Największą zaletą Cythona jest możliwość korzystania z języka Cython będącego nadzbiorem składni Pythona. Umożliwia on także tworzenie rozszerzeń z czystego kodu Pythona, wykorzystując kompilację typu *źródła-do-źródeł*. Jest to najprostszy sposób wykorzystania Cythona, ponieważ niemal nie wymaga modyfikacji kodu, a pozwala bardzo małym kosztem błyskawicznie uzyskać znaczne przyspieszenie wykonywania aplikacji.

Cython udostępnia funkcję pomocniczą `cythonize` pozwalającą na prostą integrację procesu kompilacji rozszerzeń z narzędziami dystrybucji `distutils` i `setuptools`. Załóżmy, że chcemy skompilować z użyciem Cythona naszą wzorcową implementację funkcji `fibonacci()` zapisanej w czystym Pythonie. Jeśli znajduje się ona w module `fibonacci`, to minimalna zawartość skryptu *setup.py* będzie wyglądać następująco:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    name='fibonacci',
    ext_modules=cythonize(['fibonacci.py'])
)
```

Cython wykorzystywany jako narzędzie kompilacji kodu Pythona ma dodatkową zaletę. Kompilacja modułów w trybie *źródła-do-źródeł* może być całkowicie opcjonalną częścią procesu instalacyjnego dystrybuowanego pakietu. Jeśli docelowe środowisko uruchomieniowe nie ma zainstalowanego Cythona lub dowolnego elementu zależności wymaganych do kompilacji, pakiet może być zainstalowany jako *tradycyjny* pakiet Pythona. Użytkownik nie powinien zauważyć żadnych różnic w działaniu kodu dystrybuowanego w ten sposób.

Częstą praktyką dystrybucyjną rozszerzeń utworzonych z użyciem Cythona jest umieszczanie w archiwum dystrybucyjnym zarówno źródeł Cythona/Pythona, jak i wygenerowanego z nich kodu języka C. Dzięki temu pakiet może być zainstalowany na trzy sposoby, w zależności od tego, jakie elementy procesu budowy są dostępne w docelowym środowisku instalacyjnym:

- Jeśli w docelowym środowisku dostępny jest Cython, kod C rozszerzenia będzie wygenerowany z kodu Pythona/Cythona dostępnego w archiwum dystrybucji.
- Jeśli Cython nie jest dostępny, ale środowisko zawiera pozostałe zależności wymagane do kompilacji rozszerzenia (kompilator C, pliki nagłówkowe interfejsu Python/C), rozszerzenie będzie skompilowane z dostarczonych plików źródłowych C dostępnych w archiwum dystrybucji.

- Jeśli żadne z wymienionych zależności kompilacyjnych nie są dostępne, ale rozszerzenie miało być generowane z czystych źródeł Pythona, kompilacja będzie pominięta, a kod zostanie zainstalowany jak normalny pakiet Pythona.

Załączanie w archiwach dystrybucyjnych zarówno kodu podstawowego, jak i wstępnie wygenerowanego kodu C jest praktyką rekomendowaną przez oficjalną dokumentację Cythona. Ta sama dokumentacja sugeruje, aby kompilacja z użyciem Cythona była opcjonalna i domyślnie wyłączona, ponieważ docelowe środowisko uruchomieniowe, jakim posługuje się użytkownik, może posiadać niekompatybilną wersję Cythona. Przy tak dużej popularności narzędzi do izolacji środowisk wydaje się to jednak coraz mniej znaczącym problemem. W dodatku Cython jest poprawnym pakietem Pythona dostępnym w repozytorium PyPI, może być więc z łatwością oznaczany jako jedna z zależności pakietu wymagana w konkretnej wersji. Decyzja o takiej dystrybucji pakietu nie powinna być jednak podejmowana pochopnie, ponieważ może się to wiązać z poważnymi konsekwencjami. Dużo bezpieczniej jest wykorzystać opcję `extras_require` modułu `setuptools` i pozostawić decyzję o instalacji z użyciem Cythona użytkownikowi za pomocą zmiennych środowiskowych:

```
import os

from distutils.core import setup
from distutils.extension import Extension

try:
    # Kompilacja Cythonem odbędzie się tylko,
    # jeśli Cython jest dostępny...
    import Cython
    # ...oraz gdy odpowiednia zmienna środowiskowa
    # jawnie określa wymaganie takowej kompilacji.
    USE_CYTHON = bool(os.environ.get("USE_CYTHON"))

except ImportError:
    USE_CYTHON = False

ext = '.pyx' if USE_CYTHON else '.c'

extensions = [Extension("fibonacci", ["fibonacci"+ext])]

if USE_CYTHON:
    from Cython.Build import cythonize
    extensions = cythonize(extensions)

setup(
    name='fibonacci',
    ext_modules=extensions,
    extras_require={
        # Cython będzie wskazany w tej konkretnej
        # wersji jako zależność pakietu, jeśli
```

```
            # pakiet zostanie zainstalowany w opcji
            # [with-cython].
            'with-cython': ['cython==0.23.4']
        }
)
```

Narzędzie `pip` pozwala na instalację pakietów z opcjonalnymi zestawami zależności `extas_require` poprzez dodanie do nazwy instalowanego pakietu końcówki [*`nazwa-opcji`*]. Dla przedstawionego powyżej przykładu skryptu *setup.py* instalacja Cythona i lokalna kompilacja z jego użyciem może być wykonana za pomocą następującego polecenia:

```
$ USE_CYTHON=1 pip install .[with-cython]
```

Cython jako język

Cython jest kompilatorem, ale także nadzbiorem języka Python. Nadzbiór języka oznacza, że każdy poprawny kod Pythona jest poprawnym kodem Cythona, ale dozwolone są także pewne dodatkowe elementy składni. Dodatkowe elementy składni w Cythonie mają na celu ułatwienie wywoływania funkcji języka C oraz definiowanie statycznych typów (zarówno Pythona, jak i C) jako zmiennych i atrybutów klas. Dlatego każdy kod napisany w Pythonie jest jednocześnie poprawnym kodem napisanym w Cythonie. To dzięki temu tak łatwo jest dokonać kompilacji zwykłych modułów Pythona do postaci rozszerzeń C za pomocą kompilatora Cython.

Nie poprzestaniemy jednak na samym stwierdzeniu, że kod Pythona jest jednocześnie kodem Cythona. Postaramy się usprawnić definicję naszego modułu, wykorzystując unikalne elementy języka Cython. Nie będą to znaczne optymalizacje, a jedynie poprawki pozwalające funkcji `fibonacci()` na wykorzystanie zalet Cythona bez zmieniania jej ogólnych założeń.

Źródła Cythona wykorzystują własne rozszerzenia plików: *.pyx* (zamiast tradycyjnego *.py*). Załóżmy, że wciąż zamierzamy zaimplementować funkcję zwracającą wyrazy ciągu Fibonacciego. Zawartość pliku *fibonacci.pyx* mogłaby być następująca:

```
"""Moduł Cythona udostępniający funkcję fibonacci."""


def fibonacci(unsigned int n):
  """Zwróć n-ty wyraz ciągu Fibonacciego wyznaczony rekurencyjnie."""
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Jak widzisz, jedyną rzeczą, która na chwilę obecną uległa zmianie, jest sygnatura funkcji `fibonacci()`. Dzięki opcjonalnemu statycznemu typowaniu Cythona możemy zadeklarować argument `n` jako liczbę całkowitą bez znaku (czyli `unsigned int`), co powinno już nieznacznie poprawić działanie naszej funkcji. Dodatkowo tak utworzona funkcja zapewnia lepszą obsługę argumentów niż nasze rozszerzenie napisane od podstaw w C. Jeśli argument funkcji Cythona

posiada zdefiniowany typ, to skompilowane rozszerzenie samo zajmie się rzutowaniem typów oraz zgłaszaniem błędów przepełnienia za pomocą odpowiednich rozszerzeń:

```
>>> from fibonacci import fibonacci
>>> fibonacci(5)
5
>>> fibonacci(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 21, in fibonacci.fibonacci (fibonacci.c:704)
OverflowError: can't convert negative value to unsigned int
>>> fibonacci(10 ** 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fibonacci.pyx", line 21, in fibonacci.fibonacci (fibonacci.c:704)
OverflowError: value too large to convert to unsigned int
```

Wiemy już, że Cython kompiluje jedynie źródła Pythona/Cythona do kodu C (a nie postaci binarnej) oraz że wygenerowany kod korzysta z tego samego interfejsu Python/C co zwykłe rozszerzenia w C pisane ręcznie. Zwróć uwagę na to, że funkcja `fibonacci()` jest funkcją rekurencyjną, co oznacza, że bardzo często będzie wywoływać samą siebie. Mimo że zdefiniowaliśmy ją jako funkcję z argumentem o statycznym typie, podczas rekurencyjnego wywołania będzie traktowana jak każda inna funkcja Pythona. Wartości `n-1` oraz `n-2` zostaną ponownie zapakowane do obiektów Pythona i znów rozpakowane wewnątrz wywołania rekurencyjnego. Proces ten będzie powtarzany wielokrotnie, póki nie zostanie osiągnięty ostatni poziom rekurencji. Nie jest to wielkim problemem, ale zużywa dużo więcej zasobów niż to konieczne.

Możemy ograniczyć narzut związany z wywoływaniem funkcji Pythona i przetwarzaniem jej argumentów poprzez oddelegowanie głównej części naszej funkcji do osobnej funkcji C, która w ogóle nie będzie miała do czynienia ze strukturami Pythona. Robiliśmy tak już wcześniej, pisząc rozszerzenie w C, i możemy zrobić to również w Cythonie. Do definicji funkcji w stylu C służy słowo kluczowe `cdef`. Funkcje zadeklarowane z użyciem `cdef` mogą przyjmować i zwracać jedynie typy języka C:

```
cdef long long fibonacci_cc(unsigned int n):
    if n < 2:
        return n
    else:
        return fibonacci_cc(n - 1) + fibonacci_cc(n - 2)


def fibonacci(unsigned int n):
  """Zwróć n-ty wyraz ciągu Fibonacciego wyznaczony rekurencyjnie."""
    return fibonacci_cc(n)
```

Możemy nawet pójść dalej. Przy rozszerzeniu napisanym w C ostatecznie pokazaliśmy, jak zwalniać globalną blokadę interpretera (GIL) podczas wywoływania funkcji języka C, sprawiając, że rozszerzenie jest odrobinę bardziej przyjazne dla aplikacji wielowątkowych. W poprzednich

przykładach korzystaliśmy w tym celu z makr preprocesora `Py_BEGIN_ALLOW_THREADS` oraz `Py_END_ALLOW_THREADS`. Składnia Cythona jest prostsza w obsłudze i łatwiejsza do zapamiętania. Mechanizm GIL może być zwolniony wokół sekcji kodu za pomocą instrukcji `with nogil`:

```
def fibonacci(unsigned int n):
  """ Zwróć n-ty wyraz ciągu Fibonacciego wyznaczony rekurencyjnie. """
    with nogil:
        result = fibonacci_cc(n)

    return fibonacci_cc(n)
```

Możesz także oznaczyć całą funkcję C jako bezpieczną do wywołania poza mechanizmem GIL:

```
cdef long long fibonacci_cc(unsigned int n) nogil:
    if n < 2:
        return n
    else:
        return fibonacci_cc(n - 1) + fibonacci_cc(n - 2)
```

Należy pamiętać, że funkcje takie nie mogą przyjmować jako argumentów ani zwracać obiektów Pythona. Za każdym razem, gdy funkcja oznaczona kwalifikatorem `nogil` musi wywołać funkcję interfejsu Python/C lub skorzystać ze struktur Pythona, musi jawnie założyć blokadę interpretera za pomocą instrukcji `with gil`.

Wyzwania związane z rozszerzeniami

Będę szczery i przyznam się, że rozpocząłem moją przygodę z Pythonem tylko dlatego, że byłem zmęczony złożonością tworzenia oprogramowania w C i C++. Wielu programistów zaczyna naukę Pythona, ponieważ stwierdzają, że języki, z których korzystali dotychczas, nie spełniają ich potrzeb. Programowanie w Pythonie w porównaniu do C, C++ czy Javy to istna przyjemność. Wszystko zdaje się proste i doskonale zaprojektowane. Wydaje się, że nie ma problemu, którego nie dałoby się rozwiązać, oraz że nie są nam już potrzebne żadne inne języki programowania.

Nic bardziej mylnego. Owszem, Python jest niezwykłym językiem z mnóstwem wspaniałych funkcjonalności i może być z powodzeniem wykorzystywany w wielu dziedzinach. Nie oznacza to jednak, że jest perfekcyjny i pozbawiony wad. Łatwo pisać w nim zrozumiały kod, ale ta łatwość ma swoją cenę. Python nie jest tak wolny, jak się wydaje wielu sceptykom, ale nigdy też nie będzie tak szybki jak C. Język jest z definicji bardzo przenośny, ale nie oznacza to, że jego interpreter jest dostępny na każdej możliwej platformie. Listę tego typu zastrzeżeń można rozwijać niemal bez końca.

Jednym z rozwiązań wielu powyższych problemów są rozszerzenia interpretera, dzięki którym możemy skorzystać z zalet *starego dobrego* C bezpośrednio w interpreterze Pythona. W tym momencie pojawia się pytanie: czy naprawdę korzystamy z Pythona po to, aby rozszerzać go za pomocą C lub dowolnego innego języka? Odpowiedź brzmi oczywiście „nie”. Rozszerzenia są wyłącznie niewygodną koniecznością w sytuacji, gdy nie mamy żadnego innego wyboru.

Dodatkowa złożoność

Nie jest tajemnicą, że rozwijanie aplikacji w wielu językach programowania jednocześnie to niełatwe zadanie. Python i C to dwie całkowicie różne technologie i niezwykle trudno jest wyszukać pomiędzy nimi jakiekolwiek cechy wspólne. Smutną prawdą jest także to, że nie sposób znaleźć jakąkolwiek aplikację wolną od błędów. Jeśli rozszerzenia staną się znaczącą częścią Twojej bazy kodów, debugowanie może zrobić się niezwykle złożone. Nie tylko dlatego, że debugowanie kodu w C wymaga zupełnie innego sposobu pracy i zestawu narzędzi, ale również dlatego, że zmusza do ciągłej zmiany kontekstu.

Wszyscy jesteśmy ludźmi i mamy ograniczone zdolności poznawcze. Są oczywiście ludzie mogący wydajnie pracować jednocześnie na wielu poziomach abstrakcji i z wykorzystaniem rozmaitych stosów technologicznych, ale zdają się wyjątkowo rzadkim gatunkiem. Bez względu na to, jak zdolnym programistą jesteś, utrzymywanie hybrydowych rozwiązań ma dodatkową cenę. Będzie to większy koszt związany z ciągłym przeskakiwaniem pomiędzy różnymi językami bądź też dodatkowy stres, który w końcu uczyni Cię mniej efektywnym.

Zgodnie z danymi indeksu TIOBE język C od lat jest jednym z najpopularniejszych języków programowania. Mimo to wciąż wielu programistów Pythona wie bardzo niewiele na jego temat. Uważam, że język C powinien być *lingua franca* w świecie programistów. Niestety, jest mało prawdopodobne, by moje zdanie zmieniło coś w tym temacie. Python jest tak uwodzącym i prostym w nauce językiem, że wielu programistów szybko zapomina o wszystkich swoich wcześniejszych doświadczeniach i przerzuca się na nową technologię. Ale programowanie to nie jazda na rowerze. Umiejętności programistyczne zanikają wyjątkowo szybko, gdy nie są używane i doskonalone. Nawet bardzo doświadczeni programiści C ryzykują stopniową utratę swoich umiejętności, zbyt długo nurkując w Pythonie. Wszystko to zmierza do ostatecznej konkluzji: niezwykle trudno jest znaleźć programistów, który będą mogli sprawnie rozwijać Twoje rozszerzenia. W projektach *open source* oznacza to mniej dobrowolnych współpracowników. W projektach z zamkniętymi źródłami oznacza to zaś, że nie wszyscy Twoi koledzy z pracy będą potrafili rozwijać kod bez wprowadzania błędów.

Debugowanie

Jeśli chodzi o błędy, to te występujące w rozszerzeniach są z reguły katastrofalne w skutkach. Niewątpliwą zaletą statycznego typowania jest możliwość wychwycenia wielu błędów już na etapie kompilacji (w dodatku takich błędów, których znalezienie wymagałoby bardzo rygorystycznego testowania oraz pełnego pokrycia testami). Z drugiej strony, wymagane jest ręczne zarządzanie pamięcią, a to jedna z głównych przyczyn większości błędów spotykanych w kodzie C. W najlepszym przypadku błędy zarządzania pamięcią będą skutkować tylko wyciekiem pamięci, który stopniowo pochłonie wszystkie dostępne zasoby systemowe. Najlepszy przypadek wcale nie jest tym, z którym uporać się będzie najłatwiej. Wycieki pamięci są wyjątkowo trudne w identyfikacji i odnajdywaniu bez użycia wyspecjalizowanych zewnętrznych narzędzi typu Valgrind. I tak w większości przypadków błędy zarządzania pamięcią poskutkują błędami naruszenia pamięci, których nie da się w elegancki sposób obsłużyć w Pythonie i które spowodują zakończenie pracy

interpretera bez podniesienia jakiegokolwiek wyjątku. W ostateczności będziesz musiał się uzbroić w narzędzia, z których korzysta bardzo niewielu programistów Pythona. Zwiększa to niewątpliwie złożoność środowiska pracy oraz procesu rozwoju oprogramowania.

Korzystanie z dynamicznych bibliotek bez pisania rozszerzeń

Dzięki `ctypes` (moduł standardowej biblioteki Pythona) oraz `cffi` (zewnętrzny pakiet) możesz dokonać integracji dowolnej dynamicznej/współdzielonej biblioteki bez względu na to, w jakim języku została napisana. W dodatku możesz to zrobić, posługując się czystym kodem Pythona. Jest to więc ciekawa alternatywa dla rozszerzeń pisanych w języku C.

Nie oznacza to, że niepotrzebna jest Ci jakakolwiek wiedza o C. Oba rozwiązania wymagają od Ciebie znacznego zrozumienia C i ogólnych zasad działania dynamicznych bibliotek. Z drugiej strony, zdejmują z Twoich barków ciężar ręcznego liczenia referencji oraz ograniczają ryzyko popełnienia bolesnych błędów zarządzania pamięcią. W dodatku korzystanie z dynamicznych bibliotek za pomocą `ctypes` lub `cffi` powinno być bardziej przenośne niż pisanie i kompilacja rozszerzeń w C.

ctypes

`ctypes` jest najpopularniejszym modułem do wywoływania funkcji z dynamicznych bądź współdzielonych bibliotek bez potrzeby pisania własnych rozszerzeń. Stanowi część standardowej biblioteki Pythona, więc jest dostępny zawsze i nie wymaga instalowania żadnych zewnętrznych zależności. Jest to **interfejs funkcji obcych** (ang. *Foreign Function Interface* — FFI) udostępniający dodatkowo odpowiedni API do tworzenia typów danych kompatybilnych z C.

Ładowanie bibliotek

`ctypes` udostępnia cztery klasy ładujące dynamiczne biblioteki oraz dwie konwencje korzystania z nich. Klasy reprezentujące dynamiczne i współdzielone biblioteki to `ctypes.CDLL`, `ctypes.PyDLL`, `ctypes.OleDLL` oraz `ctypes.WinDLL`. Dwie ostatnie dostępne są tylko w systemie Windows, więc nie będę się nad nimi zbytnio rozwodził. Różnice pomiędzy `CDLL` a `PyDLL` są następujące:

- `ctypes.CDLL` — ta klasa reprezentuje załadowane współdzielone biblioteki. Funkcje w tych bibliotekach wykorzystują standardową konwencję wywołania i oczekuje się od nich, by zwracały wartości typu `int`. GIL jest zwalniany podczas ładowania biblioteki.
- `Ctypes.PyDLL` — ta klasa działa jak CDLL, z tą różnicą, że GIL nie jest zwalniany podczas ładowania biblioteki. Po wykonaniu sprawdzana jest flaga błędów Pythona i ewentualnie podnoszony odpowiedni wyjątek. Klasa ta jest użyteczna jedynie przy wywoływaniu funkcji interfejsu Python/C.

Aby załadować bibliotekę, możesz bezpośrednio utworzyć instancję jednej z powyższych klas (przekazując właściwe parametry) lub też skorzystać z wywołania `LoadCall()` z podmodułu powiązanego z odpowiednią klasą:

- `ctypes.cdll.LoadLibrary()` dla `ctypes.CDLL`;
- `ctypes.pydll.LoadLibrary()` dla `ctypes.PyDLL`;
- `ctypes.windll.LoadLibrary()` dla `ctypes.WinDLL`;
- `ctypes.oledll.LoadLibrary()` dla `ctypes.OleDLL`.

Największym wyzwaniem podczas ładowania współdzielonych bibliotek jest odnajdywanie ich w przenośny i niezawodny sposób. Różne systemy operacyjne wykorzystują różne rozszerzenia dla bibliotek współdzielonych (*.dll* w systemie Windows, *.dylib* w macOS, *.so* w Linuksie) i przechowują je w różnych miejscach. Największym grzesznikiem w tej dziedzinie jest Windows, w którym nie ma ustandaryzowanego systemu organizacji bibliotek. Z tego powodu nie będę się zagłębiał w ładowanie bibliotek współdzielonych w Windowsie i skupię się na systemach Linux i macOS, które rozwiązują ten problem w konsekwentny i podobny sposób. Jeśli mimo wszystko jesteś zainteresowany ładowaniem bibliotek w systemie Windows, zapoznaj się z oficjalną dokumentacją `ctypes`, która zawiera mnóstwo informacji na ten temat (*https://docs.python.org/3.5/library/ctypes.html*).

Obie konwencje ładowania bibliotek (funkcja `LoadLibrary()` oraz konkretne klasy) wymagają użycia pełnej nazwy biblioteki. Oznacza to, że należy korzystać z odpowiednich dla danego systemu operacyjnego prefiksów i sufiksów. Na przykład aby załadować standardową bibliotekę C w systemie Linux, należy zastosować następujący kod:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.so.6')
<CDLL 'libc.so.6', handle 7f0603e5f000 at 7f0603d4cbd0>
```

W systemie macOS wygląda to następująco:

```
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('libc.dylib')
```

Na szczęście podmoduł `ctypes.util` dostarcza funkcję pomocniczą `find_library()` pozwalającą wyszukać bibliotekę bez konieczności podawania jej prefiksów i sufiksów. Funkcja ta działa w każdym systemie operacyjnym posiadającym jasno określony schemat nazywania i przechowywania bibliotek współdzielonych:

```
>>> import ctypes
>>> from ctypes.util import find_library
>>> ctypes.cdll.LoadLibrary(find_library('c'))
<CDLL '/usr/lib/libc.dylib', handle 7fff69b97c98 at 0x101b73ac8>
>>> ctypes.cdll.LoadLibrary(find_library('bz2'))
<CDLL '/usr/lib/libbz2.dylib', handle 10042d170 at 0x101b6ee80>
>>> ctypes.cdll.LoadLibrary(find_library('AGL'))
<CDLL '/System/Library/Frameworks/AGL.framework/AGL',
↪handle 101811610 at 0x101b73a58>
```

Wywoływanie funkcji C za pomocą ctypes

Powszechną praktyką po załadowaniu biblioteki jest przechowanie jej obiektu jako zmiennej w przestrzeni nazw modułu o nazwie takiej samej jak załadowana właśnie biblioteka. Funkcje biblioteki są dostępne jako atrybuty tego obiektu, a więc wywoływanie wizualnie niczym się nie różni od wywoływania funkcji dowolnego modułu Pythona:

```
>>> import ctypes
>>> from ctypes.util import find_library
>>>
>>> libc = ctypes.cdll.LoadLibrary(find_library('c'))
>>>
>>> libc.printf(b"Hello world!\n")
Hello world!
13
```

Niestety, żaden z wbudowanych typów Pythona (z wyjątkiem `int`, `str` oraz `bytes`) nie jest kompatybilny z typami C. W związku z tym większość obiektów przed przekazaniem jako atrybuty funkcji musi być owinięta za pomocą odpowiednich klas udostępnionych w module `ctypes`. Poniższa tabela, pochodząca z oficjalnej dokumentacji `ctypes`, zawiera pełną listę kompatybilnych typów:

Typ ctypes	Typ C	Typ Pythona
`c_bool`	`_Bool`	`bool`
`c_char`	`char`	Jednoznakowy obiekt `bytes`
`c_wchar`	`wchar_t`	Jednoznakowy obiekt `str`
`c_byte`	`char`	`int`
`c_ubyte`	`unsigned char`	`int`
`c_short`	`short`	`int`
`c_ushort`	`unsigned short`	`int`
`c_int`	`int`	`int`
`c_uint`	`unsigned int`	`int`
`c_long`	`long`	`int`
`c_ulong`	`unsigned long`	`int`
`c_longlong`	`__int64` lub `long long`	`int`
`c_ulonglong`	`unsigned __int64` lub `unsigned long long`	`int`
`c_size_t`	`size_t`	`int`
`c_ssize_t`	`ssize_t` lub `Py_ssize_t`	`int`
`c_float`	`float`	`float`
`c_double`	`double`	`float`
`c_longdouble`	`long double`	`float`
`c_char_p`	`char *` (zakończony znakiem `NULL`)	`bytes` lub `None`
`c_wchar_p`	`wchar_t *` (zakończony znakiem `NULL`)	`string` lub `None`
`c_void_p`	`void *`	`int` lub `None`

Powyższa tabela nie zawiera żadnego typu, który odpowiadałby dostępnym w Pythonie podstawowym kolekcjom (krotkom i słownikom). Rekomendowanym sposobem tworzenia typów `ctypes` odpowiadających tablicom C jest wykorzystanie operatora mnożenia i wybranego podstawowego typu:

```
>>> import ctypes
>>> IntArray5 = ctypes.c_int * 5
>>> c_int_array = IntArray5(1, 2, 3, 4, 5)
>>> FloatArray2 = ctypes.c_float * 2
>>> c_float_array = FloatArray2(0, 3.14)
>>> c_float_array[1]
3.140000104904175
```

Przekazywanie funkcji Pythona jako wywołań zwrotnych C

Popularnym wzorcem projektowym jest oddelegowanie części pracy do dodatkowego wywołania zwrotnego dostarczonego przez użytkownika. Jedną z najbardziej znanych funkcji ze standardowej biblioteki języka C akceptującej wywołania zwrotne jest `qsort()` implementująca sortowanie algorytmem **Quicksort**. Jest oczywiście mało prawdopodobne, byś potrzebował korzystać z algorytmu Quicksort zamiast domyślnego w Pythonie algorytmu **Timsort**. Funkcja `qsort()` stanowi kanoniczny przykład wydajnej funkcji sortującej oraz API języka C stosowanego w wielu książkach programistycznych. Dlatego postaram się wykorzystać tę właśnie funkcję w celu demonstracji przekazywania funkcji Pythona jako wywołań zwrotnych dynamicznych bibliotek ładowanych za pomocą `ctypes`.

Zwykła funkcja Pythona nie będzie kompatybilna z typem oczekiwanym jako wywołanie zwrotne funkcji `qsort()`. Oto sygnatura funkcji `qsort()` z podręcznika użytkownika BSD, która zawiera również sygnaturę oczekiwanego wywołania zwrotnego (argument `compar`):

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

Aby wywołać funkcję `qsort()` z biblioteki `libc`, musisz podać:

- `base` — sortowana tablica jako wskaźnik `void*`.
- `nel` — liczba elementów jako `size_t`.
- `width` — rozmiar pojedynczego elementu tablicy jako `size_t`.
- `compar` — wskaźnik funkcji zwracającej wartość typu `int` i akceptującej dwa wskaźniki `void*`. Wskazywana funkcja ma porównywać między sobą pary sortowanych elementów.

Wiesz już z podrozdziału „Wywoływanie funkcji C za pomocą ctypes", jak zadeklarować tablicę C wybranego typu za pomocą operatora mnożenia i modułu `ctypes`. Argument `nel` powinien być typu `size_t`, a to przekłada się na zwykły typ `int`. Dlatego nie trzeba stosować żadnych dodatkowych zabiegów i można posłużyć się wyrażeniem `len(lista)`. Argument `width` możemy określić za pomocą funkcji `ctypes.sizeof()`. Ostatnią rzeczą, jaką musimy dostarczyć, jest kompatybilny wskaźnik funkcji jako argument `compar`.

Moduł ctypes zawiera funkcję wytwórczą CFUNCTYPE(), która pozwala na owinięcie dowolnej funkcji Pythona oraz reprezentowanie jej pod postacią kompatybilnego z C wskaźnika funkcji. Pierwszym argumentem jest typ zwracany przez owiniętą funkcję. Kolejnymi argumentami są typy akceptowanych przez funkcję parametrów. Definicja typu kompatybilnego z argumentem compar funkcji qsort() będzie następująca:

```
CMPFUNC = ctypes.CFUNCTYPE(
    # zwracany typ
    ctypes.c_int,
    # pierwszy argument wejściowy
    ctypes.POINTER(ctypes.c_int),
    # drugi argument wejściowy
    ctypes.POINTER(ctypes.c_int),
)
```

Funkcja CFUNCTYPE() wykorzystuje konwencję wywoływania cdecl, a więc jest kompatybilna z bibliotekami typów CDLL oraz PyDLL. Dynamiczne biblioteki ładowane w systemie Windows za pomocą klas WinDLL oraz OleDLL wykorzystują konwencję stdcall. Oznacza to, że wywołania zwrotne dla bibliotek tych typów muszą korzystać z innej funkcji wytwórczej w celu tworzenia kompatybilnych z C wskaźników funkcji. W ctypes funkcja ta nosi nazwę WINFUNCTYPE().

Na koniec załóżmy, że sortujemy losowo uporządkowaną listę liczb całkowitych. Poniższy listing przedstawia wszystko, czego do tej pory dowiedziałeś się o ctypes:

```
from random import shuffle

import ctypes
from ctypes.util import find_library

libc = ctypes.cdll.LoadLibrary(find_library('c'))

CMPFUNC = ctypes.CFUNCTYPE(
    # zwracany typ
    ctypes.c_int,
    # pierwszy argument wejściowy
    ctypes.POINTER(ctypes.c_int),
    # drugi argument wejściowy
    ctypes.POINTER(ctypes.c_int),
)


def ctypes_int_compare(a, b):
    # Argumenty są wskaźnikami, dlatego musimy odwoływać
    # się do nich za pomocą indeksów [0].
    print(" %s cmp %s" % (a[0], b[0]))

    # Zgodnie ze specyfikacją funkcji qsort()
    # powinniśmy zwrócić:
```

```
        # * mniej niż zero, jeśli a < b;
        # * zero, jeśli a == b;
        # * więcej niż zero, jeśli a > b.
        return a[0] - b[0]


    def main():
        numbers = list(range(5))
        shuffle(numbers)
        print("pomieszane: ", numbers)

        # Utwórz nowy typ reprezentujący tablicę
        # o długości takiej samej jak lista numbers.
        NumbersArray = ctypes.c_int * len(numbers)
        # Utwórz tablicę, posługując się nowym typem.
        c_array = NumbersArray(*numbers)

        libc.qsort(
            # wskaźnik sortowanej tablicy
            c_array,
            # długość tablicy
            len(c_array),
            # rozmiar pojedynczego elementu
            ctypes.sizeof(ctypes.c_int),
            # wywołanie zwrotne (wskaźnik funkcji)
            CMPFUNC(ctypes_int_compare)
        )
        print("posortowane: ", list(c_array))


    if __name__ == "__main__":
        main()
```

Funkcja porównująca elementy zawiera dodatkową instrukcję `print()`, dzięki czemu możemy bliżej przyjrzeć się procesowi sortowania:

```
$ python ctypes_qsort.py
pomieszane: [4, 3, 0, 1, 2]
 4 cmp 3
 4 cmp 0
 3 cmp 0
 4 cmp 1
 3 cmp 1
 0 cmp 1
 4 cmp 2
 3 cmp 2
 1 cmp 2
posortowane: [0, 1, 2, 3, 4]
```

CFFI

CFFI to *interfejs funkcji obcych* stanowiący ciekawą alternatywę dla `ctypes`. Nie jest elementem standardowej biblioteki Pythona, ale jest z łatwością dostępny jako pakiet `cffi` w repozytorium PyPI. Różni się znacząco od `ctypes` tym, że kładzie większy nacisk na ponowne wykorzystanie istniejących deklaracji języka C z ogólnodostępnych plików nagłówkowych zamiast zmuszania do tworzenia rozbudowanych interfejsów będących pomostami pomiędzy kodem Pythona a ładowanymi bibliotekami. CFFI jest dużo bardziej rozbudowany niż `ctypes` i posiada możliwość kompilacji części kodu warstwy integracyjnej do postaci rozszerzeń za pomocą kompilatora C. Może być więc stosowany jako hybrydowe rozwiązanie znajdujące się gdzieś pomiędzy `ctypes` a rozszerzeniami napisanymi bezpośrednio w C.

Ponieważ jest to bardzo duży projekt, nie sposób omówić go w kilku akapitach. Z drugiej strony, szkoda byłoby niczego o nim nie powiedzieć. Pokazałem wcześniej, jak wykorzystać funkcję `qsort()` ze standardowej biblioteki języka C za pomocą modułu `ctypes`. Zatem najlepszym sposobem na wskazanie różnic pomiędzy tymi dwoma rozwiązaniami będzie zrobienie tego samego, tym razem z wykorzystaniem pakietu `cffi`. Mam nadzieję, że poniższy blok kodu będzie wart więcej niż kilka dodatkowych akapitów tekstu:

```
from random import shuffle

from cffi import FFI

ffi = FFI()

ffi.cdef("""
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
""")
C = ffi.dlopen(None)


@ffi.callback("int(void*, void*)")
def cffi_int_compare(a, b):
    # Sygnatura wywołania zwrotnego wymaga dokładnego
    # dopasowania typów. Wykorzystuje to zdecydowanie
    # dużo mniej magii niż wywołania w ctypes, ale z
    # drugiej strony zmusza do większej dokładności.
    int_a = ffi.cast('int*', a)[0]
    int_b = ffi.cast('int*', b)[0]
    print(" %s cmp %s" % (int_a, int_b))

    # Zgodnie ze specyfikacją funkcji qsort()
    # powinniśmy zwrócić:
    # * mniej niż zero, jeśli a < b;
    # * zero, jeśli a == b;
    # * więcej niż zero, jeśli a > b.
    return int_a - int_b
```

```
def main():
    numbers = list(range(5))
    shuffle(numbers)
    print("pomieszane: ", numbers)

    c_array = ffi.new("int[]", numbers)

    C.qsort(
        # wskaźnik sortowanej tablicy
        c_array,
        # długość tablicy
        len(c_array),
        # rozmiar pojedynczego elementu
        ffi.sizeof('int'),
        # wywołanie zwrotne (wskaźnik funkcji)
        cffi_int_compare,
    )
    print("posortowane: ", list(c_array))

if __name__ == "__main__":
    main()
```

Podsumowanie

W tym rozdziale zająłem się jednym z najbardziej zaawansowanych tematów całej książki. Omówiłem zarówno powody tworzenia rozszerzeń, jak i dedykowane im narzędzia. Zacząłem od napisania pierwszego rozszerzenia z użyciem jedynie języka C oraz interfejsu Python/C. Następnie zrobiłem to samo z użyciem Cythona, aby pokazać, że tworzenie rozszerzeń może być proste, pod warunkiem że użyje się w tym celu odpowiednich narzędzi.

Wciąż istnieją powody, aby tworzyć rozszerzenia *trudniejszym sposobem* (tj. za pomocą języka C i pliku nagłówkowego *Python.h*). Mimo wszystko rekomendowanym sposobem jest korzystanie z zaawansowanych narzędzi, takich jak Cython czy Pyrex (w rozdziale wspomniany jedynie z nazwy), ponieważ czynią kod bardziej czytelnym i łatwiejszym w utrzymaniu. Zaoszczędzą Ci także większości problemów spowodowanych niedokładnym zliczaniem referencji i nieodpowiednim zarządzaniem pamięcią.

Temat rozszerzeń zakończyłem prezentacją modułu `ctypes` i pakietu `cffi` jako narzędzi alternatywnych dla integracji współdzielonych bibliotek. Ponieważ nie wymagają pisania własnych kompilowanych rozszerzeń, powinny być preferowanym rozwiązaniem w przypadku, gdy konieczne jest jedynie wywoływanie funkcji z istniejących bibliotek w formie binarnej.

Następny rozdział pozwoli Ci na chwilę odpocząć od zaawansowanych technik programistycznych. Zajmiemy się równie ważnym tematem zarządzania kodem za pomocą systemów kontroli wersji.

Skorowidz

E

F

G

H

I

J

K

R

S, Ś

Ś

T

U

V

W

Z

Profesjonalne programowanie w Pythonie

Poziom ekspert. Wydanie II

Twórcy Pythona niemal od początku starali się opracować wieloparadygmatowy język zorientowany na czytelność kodu i produktywność programisty. Dziś język ten jest uważany za wszechstronny i potężny, a do tego cechuje się prostotą i elastycznością. Nadaje się do pisania zarówno niedużych skryptów, jak i wielkich systemów, a także do wysoce specjalistycznych zadań, jak choćby analiza danych w celach naukowych. Mimo to pisanie kodu, który jest wydajny, prosty w utrzymaniu oraz łatwy w użyciu, wciąż sprawia problemy nawet zaawansowanym programistom Pythona.

Niniejsza książka to zbiór praktyk stosowanych przez najlepszych programistów pracujących z Pythonem. Jest przeznaczona dla osób zawodowo zajmujących się rozwojem oprogramowania oraz dla ambitnych pasjonatów w tej dziedzinie. Poza opisem zaawansowanych technik programowania w Pythonie znalazły się tu również informacje o narzędziach i technikach stosowanych obecnie przez profesjonalnych programistów. Opisano metody zarządzania kodem, tworzenia, dokumentowania i testowania kodu oraz zasady optymalizacji oprogramowania. Przedstawiono również wzorce projektowe, które szczególnie docenią programiści Pythona.

Python — niezawodne narzędzie dla profesjonalisty!

W książce znajdziesz:

- metodologie pracy w Pythonie i najlepsze praktyki składniowe
- rozszerzenia Pythona napisane w innych językach programowania
- techniki profilowania aplikacji
- przetwarzanie współbieżne i równoległe
- najprzydatniejsze wzorce projektowe

Michał Jaworski od blisko 10 lat programuje w Pythonie. Napisał Graceful — framework REST oparty na bibliotece Falcon. Obecnie jest architektem oprogramowania w firmie Opera Software. Poza tym projektuje wysoce wydajne rozproszone usługi sieciowe i angażuje się w liczne projekty *open source*.

Tarek Ziadé jest kierownikiem ds. technicznych w firmie Mozilla. Zajmuje się usługami sieciowymi o wielkiej skali w Pythonie na potrzeby przeglądarki Firefox. Jest także założycielem Afpy, francuskiej grupy użytkowników Pythona. Wielokrotnie był prelegentem podczas konferencji Solutions Linux, PyCon, OSCON, EuroPython i innych.

[PACKT] PUBLISHING open source* community experience distilled

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

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

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-3033-7

cena: 79,00 zł