O'REILLY®

Przewodnik po Pythonie

DOBRE PRAKTYKI I PRAKTYCZNE NARZĘDZIA

Helion

Kenneth Reitz, Tanya Schlusser

Tytuł oryginału: The Hitchhiker's Guide to Python: Best Practices for Development

Tłumaczenie: Szymon Piechaczek

ISBN: 978-83-283-3732-9

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 8.

Zarządzanie i ulepszanie kodu

W tym rozdziale omawiamy biblioteki służące do zarządzania lub upraszczania procesu rozwoju i budowy, integracji systemu, zarządzania serwerem i optymalizacji wydajności.

Ciągła integracja

Nikt nie opisuje **ciągłej integracji** (ang. *continuous integration* — CI) tak dobrze jak Martin Fowler[1]:

> Ciągła integracja to praktyka rozwijania oprogramowania, w której członkowie zespołu często integrują swoją pracę, zwykle każda osoba przynajmniej raz dziennie — co prowadzi do wielokrotnych integracji jednego dnia. Każda integracja jest weryfikowana przez automatyczne budowanie (w tym testy) w celu wykrycia błędów integracji tak szybko, jak to możliwe. Wiele zespołów przekonało się, że ten sposób prowadzi do znacznego zmniejszenia problemów z integracją i pozwala zespołom na szybszy rozwój spójnego oprogramowania.

Trzy najpopularniejsze obecnie narzędzia do CI to Travis-CI, Jenkins i Buildbot, które są opisane w kolejnych sekcjach. Są one często używane z Toxem, narzędziem Pythona do zarządzania środowiskami wirtualnymi i testami z wiersza poleceń. Travis jest przeznaczony dla wielu interpreterów Pythona na jednej platformie, a Jenkins (najpopularniejszy) i Buildbot (napisany w Pythonie) mogą zarządzać budową na wielu maszynach. Wiele osób korzysta również z Buildouta (opisanego w sekcji „Buildout" w rozdziale 3.) i z Dockera (opisanego w sekcji „Docker" w rozdziale 3.) do szybkiego i powtarzalnego budowania złożonych środowisk do testowania.

[1] Fowler to propagator najlepszych praktyk w projektowaniu i rozwoju oprogramowania i jeden z największych propagatorów ciągłej integracji. Cytat pochodzi z jego postu na blogu poświęconym ciągłej integracji (*http://martinfowler.com/articles/continuousIntegration.html*). Był gospodarzem serii dyskusji na temat *test-driven development* (TDD) (*http://martinfowler.com/articles/is-tdd-dead/*) i jego związku z ekstremalnym programowaniem, prowadzonych z Davidem Heinemeierem Hanssonem (twórcą Ruby on Rails) i Kentem Beckiem (inicjatorem ruchu ekstremalnego programowania (XP — *https://pl.wikipedia.org/wiki/Programowanie_ekstremalne*) z CI jako jednym z jego kamieni węgielnych).

Tox

Tox (*http://tox.readthedocs.org/en/latest/*) jest narzędziem automatyzacji zapewniającym pakietowanie, testowanie i wdrażanie oprogramowania Pythona wprost z konsoli lub serwera CI. Jest generycznym menedżerem środowisk wirtualnych i narzędziem konsolowym do testowania, które zapewnia poniższe funkcje:

- Sprawdza, czy pakiety instalują się prawidłowo dla różnych wersji Pythona i interpreterów.
- Przeprowadza testy w każdym środowisku, konfigurując wybrane narzędzie testujące.
- Zachowuje się jak frontend do serwerów ciągłej integracji, zmniejszając zbędny kod i łącząc CI z testowaniem opartym na powłoce systemowej.

Zainstaluj go, korzystając z narzędzia `pip`:

```
$ pip install tox
```

Zarządzanie systemem

Narzędzia opisane w tym podrozdziale służą do zarządzania systemami i ich monitorowania — automatyzacji serwerów, monitorowania systemów i zarządzania przepływem pracy.

Travis-CI

Travis-CI (*https://travis-ci.org/*) jest dystrybuowanym serwerem CI, który buduje testy do projektów z otwartym źródłem za darmo. Zapewnia mechanizmy pozwalające na przeprowadzanie automatycznych testów Pythona i bezproblemowo integruje się z GitHubem. Możesz nawet ustawić go do komentowania swoich *pull requestów*[2], niezależnie od tego, czy zbiór zmian psuje kod, czy nie. Jeśli zatem hostujesz swój kod na GitHubie, Travis-CI jest świetnym i łatwym sposobem na rozpoczęcie ciągłej integracji. Travis-CI może budować Twój kod na wirtualnej maszynie, na której działa Linux, macOS lub iOS.

Aby rozpocząć, dodaj do swojego repozytorium plik *.travis.yml* z przykładową zawartością:

```
language: python
python:
    - "2.6"
    - "2.7"
    - "3.3"
    - "3.4"
script: python tests/test_all_of_the_units.py
branches:
only:
    - master
```

To przetestuje Twój projekt na wszystkich wypisanych wersjach Pythona przez włączenie podanego skryptu i zbuduje tylko *branch* `master`. Istnieje znacznie więcej opcji, które możesz tu dołączyć, jak powiadomienia przed krokami i po krokach itp. Wszystkie te opcje opisuje bardzo dokładnie dokumentacja Travis-CI (*http://about.travis-ci.org/docs/*). Aby korzystać z Toxa razem z Travis-CI, dodaj skrypt Toxa do swojego repozytorium i zmień linię ze `script:` na:

[2] Na GitHubie użytkownicy wysyłają tzw. *pull requesty*, aby powiadomić właścicieli innego repozytorium, że przygotowali zmiany, które chcieliby połączyć z głównym kodem.

```
install:
    - pip install tox
script:
    - tox
```

Aby aktywować testowanie swojego projektu, wejdź na stronę Travis-CI (*https://travis-ci.org/*) i zaloguj się za pomocą swojego konta na GitHubie. Następnie aktywuj swój projekt w ustawieniach profilu. Od teraz Twoje testy projektu będą przeprowadzane przy każdym przesłaniu na GitHuba.

Jenkins

Jenkins (*http://jenkins.io*) jest elastycznym silnikiem ciągłej integracji i obecnie najpopularniejszym silnikiem CI. Działa w Windowsie, Linuksie i macOS i jest wtyczką dla „każdego narzędzia do zarządzania kodem źródłowym, jakie istnieje". Jenkins jest serwletem Javy (odpowiednikiem aplikacji WSGI Pythona w Javie), który ma w zestawie własny kontener serwletów, więc możesz uruchomić go bezpośrednio, używając `java --jar jenkins.war`. Aby dowiedzieć się więcej, przeczytaj instrukcję instalacji Jenkinsa (*https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins*). Na stronie Ubuntu znajdziesz instrukcję, jak ustawić Jenkinsa za odwróconym proxy Apache lub Nginx.

W interakcję z Jenkinsem wchodzi się przez webowy kokpit lub jego oparte na HTTP REST-owe API[3] (np. na *http://myServer:8080/api*), co oznacza, że możemy korzystać z HTTP do komunikacji z serwerem Jenkinsa ze zdalnych maszyn. Dla przykładu spójrz na kokpit Jenkinsa w Apache, znajdujący się pod adresem *https://builds.apache.org/*.

Najczęściej używanym narzędziem Pythona do interakcji z API Jenkinsa jest python-jenkins (*https://pypi.python.org/pypi/python-jenkins*), stworzony przez zespół infrastruktury OpenStack (*https://www.openstack.org/*)[4]. Większość użytkowników Pythona konfiguruje Jenkinsa do włączania skryptu Toxa jako część procesu budowy. Aby dowiedzieć się więcej, przejrzyj dokumentację korzystania z Toxa z Jenkinsem (*http://tox.readthedocs.io/en/latest/example/jenkins.html*) oraz poradnik ustawiania Jenkinsa z wieloma maszynami budowy (*http://tinyurl.com/jenkins-setup-master-slave*).

Buildbot

Buildbot (*http://docs.buildbot.net/current/*) jest systemem Pythona do automatyzacji cyklu kompilacji (testowania) pod kątem zatwierdzania zmian w kodzie. Przypomina Jenkinsa pod tym względem, że bada Twojego menedżera kontroli źródła, szukając zmian, buduje i testuje Twój kod na wielu komputerach zgodnie z Twoimi instrukcjami (z wbudowaną obsługą Toxa) oraz komunikuje, co się stało. Działa na serwerze webowym Twisted. Przykładem tego, jak wygląda interfejs webowy, jest kokpit buildbota Chromium (*https://build.chromium.org/p/chromium/waterfall* — Chromium napędza wyszukiwarkę Chrome).

---

[3] REST oznacza *representational state transfer* — zmianę stanu poprzez reprezentacje. Nie jest to standard czy protokół, tylko zbiór zasad projektowych, rozwiniętych podczas tworzenia standardu HTTP 1.1. Lista ważnych ograniczeń architektury dla REST jest dostępna w Wikipedii (*https://en.wikipedia.org/wiki/Representational_state_transfer - Architectural_constraints*).

[4] OpenStack zapewnia darmowe oprogramowanie do obsługi chmur obliczeniowych, w tym przechowywania i wykonywania obliczeń, tak aby organizacje mogły hostować prywatne chmury na własny użytek lub publiczne chmury, za korzystanie z których mogłyby pobierać opłaty.

Ponieważ Buildout jest czysto pythonowy, jest instalowany poleceniem `pip`:

```
$ pip install buildbot
```

Wersja 0.9 ma REST-owe API (*http://docs.buildbot.net/latest/developer/apis.html*), ale wciąż pozostaje w wersji beta, więc nie będziesz mógł z niej korzystać, o ile nie sprecyzujesz wersji (np. `pip install buildout==0.9.00.9.0rc1`). Buildout uważany jest za najpotężniejsze, ale także najbardziej złożone narzędzie ciągłej integracji. Aby rozpocząć, skorzystaj ze znakomitego samouczka ze strony *http://docs.buildbot.net/current/tutorial*.

Automatyzacja serwera

Salt, Ansible, Puppet, Chef i CFEngine to narzędzia automatyzacji serwera, zapewniające administratorom systemów elegancki sposób zarządzania ich armią fizycznych i wirtualnych maszyn. Wszystkie mogą zarządzać maszynami z Linuksem, systemami uniksopodobnymi i Windowsem. My preferujemy oczywiście Salt i Ansible, które są napisane w Pythonie, ale są jeszcze ciągle nowe i częściej stosowane są inne narzędzia. W kolejnych sekcjach omówimy pokrótce wszystkie wymienione narzędzia.

Ludzie związani z Dockerem spodziewają się, że narzędzia automatyzacji systemów, takie jak Salt czy Ansible, zostaną przez Dockera *uzupełnione*, a nie *zastąpione* — więcej o tym, jak Docker pasuje do reszty DevOps, przeczytasz na stronie *http://stackshare.io/posts/how-docker-fits-into--the-current-devops-landscape*.

Salt

Salt (*http://saltstack.org/*) określa swój główny węzeł jako *master*, a węzły agentów jako *miniony* (lub *minion hosts*). Jego głównym założeniem projektowym jest szybkość — przesyłanie danych jest domyślnie realizowane za pomocą ZeroMQ, z połączeniami TCP pomiędzy masterem i jego „minionami", a członkowie zespołu Salta napisali własny (opcjonalny) protokół transmisji, RAET (*https://github.com/saltstack/raet*), który jest szybszy od TCP i nie tak stratny jak UDP.

Salt obsługuje Pythona 2.6 i 2.7. Może być zainstalowany przez narzędzie `pip`:

```
$ pip install salt # Jeszcze nie Python 3 …
```

Po konfiguracji głównego serwera i dowolnej liczby podrzędnych hostów możemy wykonywać arbitralne polecenia powłoki systemowej lub korzystać z prefabrykowanych modułów złożonych komend na własnych *minionach*. Poniższa komenda wypisuje wszystkie dostępne *miniony*, korzystając z polecenia `ping` z modułu *test* Salta:

```
$ salt '*' test.ping
```

Możesz filtrować *miniony* według ich ID lub poprzez system *grains* (*http://docs.saltstack.org/en/latest/topics/targeting/grains.html*), który korzysta ze statycznych informacji hosta, takich jak wersja systemu operacyjnego czy architektura CPU, do zapewnienia taksonomii hostów. Na przykład poniższa komenda korzysta z systemu *grains* do wypisania tylko dostępnych *minionów*, wykorzystujących CentOS:

```
$ salt -G 'os:CentOS' test.ping
```

Salt zapewnia też system stanów. Stany mogą być używane do konfiguracji *minionów*. Na przykład kiedy *minion* dostanie zadanie przeczytania poniższego pliku stanów, zainstaluje i uruchomi serwer Apache:

```
apache:
    pkg:
        - installed
    service:
        - running
        - enable: True
        - require:
            - pkg: apache
```

Pliki stanów mogą być napisane z wykorzystaniem YAML, rozszerzone za pomocą systemu szablonów Jinja2 lub mogą być modułami czystego Pythona. Aby dowiedzieć się więcej, przejrzyj dokumentację Salta (*http://docs.saltstack.com*).

Ansible

Największą przewagą Ansible (*http://ansible.com/*) względem innych narzędzi automatyzacji systemów jest to, że nie wymaga ono niczego (poza Pythonem) do zainstalowania na stałe na maszynach klientów. Wszystkie inne narzędzia[5] utrzymują uruchomionego na maszynie klienta demona, który komunikuje się z głównym serwerem. Pliki konfiguracyjne Ansible są w formacie YAML. *Playbooki* to dokumenty Ansible dotyczące konfiguracji, wdrażania i orkiestracji i są napisane w YAML z wykorzystaniem Jinja2 do szablonowania. Ansible obsługuje Pythona w wersji 2.6 oraz 2.7 i może być zainstalowane przez narzędzie `pip`:

```
$ pip install ansible  # Jeszcze nie Python 3…
```

Ansible wymaga pliku inwentarza, opisującego hosty, do których ma dostęp. Poniższy kod to przykład hosta i playbooka, który będzie pingował wszystkie hosty z pliku inwentarza. Oto przykład pliku inwentarza (*hosts.yml*):

```
[server_name]
127.0.0.1
```

Oto przykład playbooka (*ping.yml*):

```
---
- hosts: all

  tasks:
    - name: ping
      action: ping
```

Aby włączyć playbooka, należy wpisać:

```
$ ansible-playbook ping.yml -i hosts.yml --ask-pass
```

Playbook Ansible będzie pingował wszystkie hosty z pliku *host.yml*. Możesz również wybrać grupy serwerów używających Ansible. Aby dowiedzieć się więcej o Ansible, przeczytaj dokumentację (*http://docs.ansible.com/*). To świetne i szczegółowe wprowadzenie stanowi także samouczek Ansible opracowany przez Servers for Hackers (*https://serversforhackers.com/an-ansible-tutorial/*).

[5] Poza Salt-SSH, alternatywną architekturą Salta, opracowaną prawdopodobnie na potrzeby tych użytkowników, którzy chcieli, aby Salt przypominał Ansible.

Puppet

Puppet (*http://puppetlabs.com*) jest napisany w Ruby i wykorzystuje do konfiguracji własny język — PuppetScript. Ma własny serwer, *Puppet Master*, który jest odpowiedzialny za koordynację węzłów nazywanych *agentami*. *Moduły* to małe, nadające się do udostępniania jednostki kodu, napisane w celu automatyzacji lub zdefiniowania stanu systemu. Puppet Forge (*https://forge.puppetlabs.com/*) to repozytorium modułów napisanych przez społeczność dla Open Source Puppet i Puppet Enterprise.

Węzły agentów wysyłają podstawowe informacje o systemie (np. system operacyjny, jądro, adres IP i nazwę hosta) do Puppet Master. Puppet Master kompiluje następnie katalog z dostarczonymi przez agenty informacjami o tym, jak każdy węzeł powinien być skonfigurowany, i wysyła go do agenta. Agent wymusza zmianę opisaną w katalogu i wysyła raport z powrotem do Puppet Mastera.

Facter jest z kolei interesującym narzędziem, które znajduje się w pakiecie z Puppetem i pobiera podstawowe informacje o systemie. Do tych informacji można się odnieść jak do zmiennych podczas pisania modułów Puppeta:

```
$ facter kernel
Linux
$
$ facter operatingsystem
Ubuntu
```

Pisanie modułów Puppeta jest dosyć proste: moduł taki tworzą razem pliki Puppet Manifest (pliki z rozszerzeniem *.pp*). Oto przykład aplikacji *Hello World* napisanej w Puppecie:

```
#-*-coding: utf-8-*-
notify { 'Hello World, this message is getting logged into the agent node':

    # Skoro nic nie jest sprecyzowane w body, tytuł źródła jest domyślnie wiadomością notyfikacji.
}
```

Oto inny przykład, obejmujący logikę systemową. Aby odnieść się do innych faktów, poprzedź nazwę zmiennej znakiem $, tak jak poniżej w przypadku `$operatingsystem`:

```
#-*-coding: utf-8-*-
notify{ 'Mac Warning':
    message => $operatingsystem ? {
        'Darwin' => 'To zdaje się być Mac.',
        default  => 'Jestem PC.',
    },
}
```

Istnieje kilka typów zasobów dla Puppeta, ale do realizacji większości zadań zarządzania konfiguracją w zupełności wystarczy paradygmat pakiet-plik-usługa. Poniższy kod Puppeta upewnia się, że pakiet OpenSSH-Server jest zainstalowany w systemie, a usługa sshd (demon serwera SSH) dostaje powiadomienia o restartowaniu systemu za każdym razem, kiedy plik konfiguracyjny *sshd* jest zmieniany:

```
#-*-coding: utf-8-*-
package { 'openssh-server':
    ensure => installed,
}

file { '/etc/ssh/sshd_config':
    source => 'puppet:///modules/sshd/sshd_config',
    owner  => 'root',
    group  => 'root',
    mode   => '640',
```

```
    notify =>  Service['sshd'], # sshd zrestartuje się po każdej edycji tego pliku

    require  => Package['openssh-server'],
}
service { 'sshd':
    ensure    => running,
    enable    => true,
    hasstatus => true,
    hasrestart=> true,
}
```

Aby dowiedzieć się więcej, przejrzyj dokumentację Puppet Labs (*http://docs.puppetlabs.com*).

Chef

Jeśli Chef (*https://www.chef.io/chef/*) to Twój wybór do konfiguracji zarządzania, do pisania kodu infrastruktury będziesz wykorzystywał głównie Ruby. Chef przypomina Puppeta, ale został zaprojektowany z odwrotną filozofią: Puppet zapewnia framework, który ułatwia rzeczy kosztem elastyczności, podczas gdy Chef nie zapewnia praktycznie żadnego frameworku — jego celem jest być bardzo elastycznym, a więc jest trudniejszy w użyciu.

Klienty Chefa działają na każdym węźle infrastruktury i regularnie komunikują się z *serwerem* Chefa, aby system pozostawał w równowadze i zawsze odzwierciedlał obecny stan. Każdy pojedynczy klient Chefa konfiguruje się sam. Takie podejście czyni Chefa skalowalną platformą automatyzacji.

Chef wykorzystuje własne „przepisy" (ang. *recipes*), zaimplementowane w „książkach kucharskich" (ang. *cookbook*). Te „książki kucharskie", które są właściwie pakietami dla wyborów infrastruktury, są przechowywane na serwerze Chefa. Przeczytaj serię artykułów DigitalOcean na temat Chefa (*http://tinyurl.com/digitalocean-chef-tutorial*), aby dowiedzieć się, jak utworzyć prosty serwer Chefa.

Skorzystaj z komendy `knife` (*https://docs.chef.io/knife.html*), aby utworzyć *cookbook*:

```
$ knife cookbook create cookbook_name
```

Dobrym punktem startowym dla początkujących w Chefie jest serwis „Getting started with Chef" Andy'ego Gale'a (*http://gettingstartedwithchef.com/first-steps-with-chef.html*). Wiele cookbooków społeczności można znaleźć na Chef Supermarket (*https://supermarket.chef.io/cookbooks*) — są one dobrymi punktami startowymi dla Twoich własnych cookbooków. Aby dowiedzieć się więcej, zapoznaj się z pełną dokumentacją Chefa (*https://docs.chef.io/*).

CFEngine

CFEngine jest lekki, ponieważ jest napisany w C. Jego głównym celem projektowym jest odporność na porażki, osiągnięta przez autonomiczne agenty operujące w dystrybuowanej (w przeciwieństwie do architektury master-klient) sieci, w której komunikacja odbywa się z użyciem modelu Promise Theory (*https://en.wikipedia.org/wiki/Promise_theory*). Jeśli chcesz płaskiej architektury zarządzanej sieci, wypróbuj ten system.

Monitorowanie systemu i zadań

Wszystkie opisane w tej sekcji biblioteki pomagają administratorom systemów w monitorowaniu pracy, ale każda ma inne zastosowanie. Psutil dostarcza do Pythona informacje, które mogą być uzyskane przez funkcje narzędziowe Uniksa, Fabric ułatwia definiowanie i wykonywanie poleceń na

grupie zdalnych hostów poprzez SSH, a Luigi umożliwia tworzenie harmonogramu i monitorowanie długodziałających procesów wsadowych, takich jak połączone komendy Hadoopa.

Psutil

Psutil (*https://pythonhosted.org/psutil/*) to międzyplatformowy (włączając Windowsa) interfejs do obsługi różnych informacji systemowych (dotyczących np. CPU, pamięci, dysków, sieci, użytkowników i procesów) — umożliwia otrzymanie w Pythonie informacji, które większość z nas pobiera za pomocą poleceń Uniksa (*https://pl.wikipedia.org/wiki/Polecenia_systemu_operacyjnego_Unix*), takich jak `top`, `ps`, `df` i `netstat`. Można go pobrać narzędziem `pip`:

```
$ pip install psutil
```

Oto przykład, który monitoruje serwer pod kątem przeciążenia (jeśli któryś z testów — sieć, CPU — zawiedzie, program wyśle maila):

```
#-*-coding: utf-8-*-
# Funkcje do pobrania zmiennych systemowych:
from psutil import cpu_percent, net_io_counters
# Funkcje do przerwy w działaniu:
from time import sleep
# Pakiet do usług mailowych:
import smtplib
import string

MAX_NET_USAGE = 400000
MAX_ATTACKS = 4
attack = 0
counter = 0
while attack <= MAX_ATTACKS:
    sleep(4)
    counter = counter + 1
    # Sprawdź użycie CPU
    if cpu_percent(interval = 1) > 70:
        attack = attack + 1
    # Sprawdź użycie sieci
    neti1 = net_io_counters()[1]
    neto1 = net_io_counters()[0]
    sleep(1)
    neti2 = net_io_counters()[1]
    neto2 = net_io_counters()[0]
    # Oblicz bajty na sekundę
    net = ((neti2+neto2) - (neti1+neto1))/2
    if net > MAX_NET_USAGE:
        attack = attack + 1
    if counter > 25:
        attack = 0
        counter = 0

    # Napisz bardzo ważnego maila, kiedy attack jest większy niż 4
    TO = "ty@twoj_email.com"
    FROM = "webmaster@twoja_domena.com"
    SUBJECT = "Twoja domena nie posiada zasobów!"
    text = "Napraw swój serwer!!"
    BODY = string.join(
             ("From: %s" %FROM,"To: %s" %TO,"Subject: %s" %SUBJECT, "",text), "\r\n")
    server = smtplib.SMTP('127.0.0.1')
    server.sendmail(FROM, [TO], BODY)
    server.quit()
```

Aby zobaczyć dobry przykład użycia Psutil, zapoznaj się z glances (*https://github.com/nicolargo/glances/*), w pełni terminalową aplikacją, która zachowuje się jak mocno rozszerzone polecenie `top` (wypisujące działające procesy w kolejności stopnia użycia CPU lub innej zdefiniowanej przez użytkownika) z możliwościami narzędzia monitorującego klient-serwer.

Fabric

Fabric (*http://docs.fabfile.org*) jest biblioteką do ułatwiania zadań administrowania systemem. Pozwala na korzystanie z SSH dla wielu hostów i wykonywanie zadań na każdym z nich. Jest to wygodne przy administracji systemami lub wdrażaniu aplikacji. Do instalacji Fabric użyj polecenia `pip`:

```
$ pip install fabric
```

Oto pełny moduł Pythona, definiujący dwa zadania Fabric — `memory_usage` i `deploy`:

```
#-*-coding: utf-8-*-
# fabfile.py
from fabric.api import cd, env, prefix, run, task

env.hosts = ['mój_serwer1', 'mój_serwer2'] # Gdzie użyć SSH

@task
def memory_usage():
    run('wolne -m')

@task
def deploy():
    with cd('/var/www/project-env/project'):
        with prefix('. ../bin/activate'):
            run('git pull')
            run('touch app.wsgi')
```

Instrukcja `with` zagnieżdża komendy, tak aby `deploy()` dla każdego hosta wyglądało tak:

```
$ ssh nazwa_hosta cd /var/ww/project-env/project && ../bin/activate && git pull
$ ssh nazwa_hosta cd /var/ww/project-env/project && ../bin/activate && \
> touch app.wsgi
```

Mając ten kod zapisany w pliku nazwanym *fabfile.py* (domyślna nazwa modułu, której szuka `fab`), możemy naszym nowym zadaniem `memory_usage` sprawdzić użycie pamięci:

```
$ fab memory_usage
[my_server1] Executing task 'memory'
[my_server1] run: free -m
[my_server1] out:              total     used    free   shared  buffers   cached
[my_server1] out: Mem:          6964     1897    5067        0      166      222
[my_server1] out: -/+ buffers/cache:     1509    5455
[my_server1] out: Swap:            0        0       0


[my_server2] Executing task 'memory'
[my_server2] run: free -m
[my_server2] out:              total     used    free     shared  buffers  cached
[my_server2] out: Mem:          1666      902     764          0      180     572
[my_server2] out: -/+ buffers/cache:      148    1517
[my_server2] out: Swap:          895        1     894
```

Możemy także wdrażać:

```
$ fab deploy
```

Dodatkowe funkcje obejmują wykonywanie równoległe, interakcję ze zdalnymi programami i grupowanie hostów. Zrozumiałe przykłady znajdują się w dokumentacji Fabric (*http://docs.fabfile.org*).

Luigi

Luigi (*https://pypi.python.org/pypi/luigi*) to narzędzie do zarządzania strumieniami, opracowane i wydane przez Spotify. Pomaga deweloperom zarządzać całym strumieniem dużych, długo uruchomionych zadań, łącząc zapytania Hive, zapytania baz danych, zadania Hadoopa, zadania pySparka i dowolne zadania, które chcesz sam zdefiniować. Nie wszystkie zadania muszą wykorzystywać *big data* — API pozwala na tworzenie dowolnych harmonogramów — ale Spotify używało tego narzędzia podczas prac nad Hadoopem, więc wszystkie jego funkcje są częścią `luigi.contrib` (*http://luigi.readthedocs.io/en/stable/api/luigi.contrib.html*). Do instalacji wykorzystaj narzędzie `pip`:

```
$ pip install luigi
```

Luigi zawiera interfejs sieciowy, więc użytkownicy mogą filtrować swoje zadania i przeglądać wykresy zależności strumienia i jego postępu. Przykłady zadań Luigi znajdziesz w repozytorium na GitHubie (*https://github.com/spotify/luigi/tree/master/examples*), ale możesz również zapoznać się z dokumentacją (*http://luigi.readthedocs.io/*).

Szybkość

Ten podrozdział przedstawia najpopularniejsze sposoby społeczności Pythona na optymalizację pod kątem szybkości działania. Tabela 8.1 pokazuje, co możesz zrobić po wykonaniu podstawowych rzeczy, takich jak profilowanie kodu (*https://docs.python.org/3.5/library/profile.html*) czy porównanie opcji dla wstawek kodu (*https://docs.python.org/3.5/library/timeit.html*) w celu wyciągnięcia tyle, ile tylko się da, bezpośrednio z Pythona.

Tabela 8.1. Opcje przyspieszania

Opcja	Licencja	Powody użycia
Threading	PSFL	• Pozwala na tworzenie wielu wątków wykonania. • Threading (podczas korzystania z CPythona z powodu GIL) nie używa wielu procesów; różne wątki przełączają się, kiedy jeden jest blokowany, co jest przydatne, kiedy spowolnienie jest spowodowane niektórymi blokującymi zadaniami, np. czekaniem na I/O. • Brak GIL w niektórych implementacjach Pythona, takich jak Jython czy IronPython.
Multiprocessing/ subprocess	PSFL	• Narzędzia w bibliotece multiprocessing pozwalają Ci wykonywać kolejne procesy Pythona, obchodząc GIL. • Subprocess pozwala na uruchomienie wielu procesów wiersza poleceń.

Tabela 8.1. Opcje przyspieszania (ciąg dalszy)

Opcja	Licencja	Powody użycia
PyPy	Licencja MIT	• Jest interpreterem Pythona (obecnie Python 2.7.10 lub 3.2.5), który zapewnia kompilowanie *just-in-time* do C, kiedy to możliwe. • Bezwysiłkowy: nie wymaga żadnego kodowania, a zazwyczaj daje duże przyspieszenie. • Jest zastępcą CPythona, który zazwyczaj działa — każda biblioteka C powinna korzystać z CFFI lub znajdować się na liście kompatybilności PyPy (*http://pypy.org/compat.html*).
Cython	Licencja Apache	• Zapewnia dwa sposoby na statyczną kompilację kodu Pythona: pierwszy to użycie języka adnotacji, Cython (**.pxd*)… • …a drugim sposobem jest statyczna kompilacja czystego Pythona i korzystanie z zapewnionych przez Cythona dekoratorów do sprecyzowania typu obiektu.
Numba	Licencja BSD	• Zapewnia zarówno kompilację statyczną (przez własne narzędzie pycc), jak i kompilację *just-in-time* do *kodu maszynowego*, który korzysta z tablic NumPy. • Wymaga Pythona 2.7 lub 3.4+, biblioteki llvmlite (*http://llvmlite.pydata.org/en/latest/install/index.html*) i zależnej od niej infrastruktury kompilatora LLVM (Low-Level Virtual Machine).
Weave	Licencja BSD	• Zapewnia naturalny sposób na „splatanie" kilku linii C z Pythonem, ale używaj go tylko, jeśli już korzystasz z Weave. • W przeciwnym wypadku skorzystaj z Cythona — Weave jest już przestarzałe.
PyCUDA/gnumpy/ TensorFlow/ Theano/PyOpenCL	MIT/zmodyfikowane BSD/BSD/BSD/MIT	• Te biblioteki zapewniają różne sposoby na użycie procesora graficznego NVIDII, o ile masz taki i potrafisz zainstalować architekturę CUDA (*http://docs.nvidia.com/cuda/*). • PyOpenCL może korzystać z procesorów innych niż te od NVIDII. • Każda ma własne zastosowanie — na przykład gnumpy ma być zamiennikiem NumPy.
Bezpośrednie korzystanie z bibliotek C/C++	—	• Poprawienie szybkości jest warte dodatkowego czasu poświęconego na programowanie w C/C++.

Mogłeś już usłyszeć o globalnej blokadzie interpretera (GIL — *http://wiki.python.org/moin/GlobalInterpreterLock*) — to sposób, w jaki implementacje C Pythona zezwalają na działanie wielu wątkom naraz. Zarządzanie pamięcią Pythona nie jest w pełni odporne na wątki, więc blokada

GIL jest wymagana, aby zapobiec wykonywaniu tego samego kodu Pythona przez wiele wątków jednocześnie.

O GIL często mówi się jak o ograniczeniu Pythona, ale nie jest to tak wielki problem — jest to niedogodność tylko wówczas, gdy procesy są związane z CPU (w tym przypadku, jak w NumPy lub bibliotekach kryptografii, które omówimy później, kod jest przepisywany w C i ujawniany w wiązaniach Pythona). W przypadku pozostałych zastosowań (jak I/O sieci lub I/O pliku) spowolnienie wynika z blokowania kodu w pojedynczym wątku podczas czekania na I/O. Możesz rozwiązać problem blokowania, używając wątków lub programowania sterowanego zdarzeniami.

Powinniśmy zaznaczyć, że w Pythonie 2 istniały wolniejsze i szybsze wersje bibliotek — String IO i cStringIO, czy też ElementTree i cElementTree. Implementacje w C są szybsze, ale muszą być specjalnie importowane. Od Pythona 3.3 podstawowe wersje importują z szybszych implementacji, kiedy tylko jest to możliwe, a biblioteki o nazwie poprzedzonej literą *c* są przestarzałe.

Autor książki *Writing Idiomatic Python* (*http://bit.ly/writing-idiomatic-python*), Jeff Knupp, napisał na blogu post o obchodzeniu GIL (*http://bit.ly/pythons-hardest-problems*), cytując głębsze przemyślenia Davida Beazleya na ten temat[6].

W kolejnych sekcjach szczegółowo omówione są threading i inne opcje optymalizacji z tabeli 8.1.

Threading

Biblioteka threading Pythona pozwala na użycie wielu wątków. Z powodu GIL (przynajmniej w CPythonie) na jednym interpreterze będzie działał tylko jeden proces, co znaczy, że wzrost wydajności nastąpi tylko wówczas, kiedy przynajmniej jeden wątek będzie blokowany (np. przez I/O). Inną opcją jest użycie obsługi zdarzeń — na ten temat dowiesz się więcej w rozdziale 9., w sekcji „Narzędzia sieciowe wydajności w standardowej bibliotece Pythona".

Kiedy masz wiele wątków, jądro zauważa, że jeden jest blokowany przez I/O, więc przełącza, aby umożliwić wykorzystanie procesora kolejnemu wątkowi, dopóki nie zostanie on zablokowany lub zakończony. Wszystko to dzieje się automatycznie, kiedy uruchomisz swoje wątki. Dobry przykład skorzystania z wątkowania znajdziesz na stronach StackOverflow (*http://bit.ly/threading-in-python*), a serwis Python Module of the Week zawiera interesujące wprowadzenie do threading (*https://pymotw.com/2/threading/*). Możesz również sprawdzić dokumentację threading w bibliotece standardowej (*https://docs.python.org/3/library/threading.html*).

Multiprocessing

Moduł multiprocessing standardowej biblioteki Pythona (*https://docs.python.org/3/library/multiprocessing.html*) zapewnia sposób na obejście GIL — włączenie dodatkowych interpreterów Pythona. Osobne procesy mogą komunikować się ze sobą, korzystając z `multiprocessing.Pipe` lub z `multiprocessing.Queue`, albo dzielić pamięć za pomocą `multiprocessing.Array` lub `multiprocessing.Value`, które

[6] David Beazley opracował świetny poradnik (PDF — *http://www.dabeaz.com/python/UnderstandingGIL.pdf*), który opisuje działanie blokady GIL. Opisuje również nową blokadę GIL (PDF — *http://www.dabeaz.com/python/NewGIL.pdf*) w Pythonie 3.2. Wyniki jego analiz pokazują, że maksymalizacja wydajności w aplikacjach Pythona wymaga dogłębnego zrozumienia GIL — tego, jak GIL wpływa na Twoje konkretne aplikacje, ile masz rdzeni oraz gdzie znajdują się punkty spowalniające Twoją aplikację.

implementują blokowanie automatycznie. Wszystkie te obiekty implementują blokowanie, aby zapobiec jednoczesnemu dostępowi przez różne procesy.

Oto przykład pokazujący, że nie zawsze zwiększenie szybkości jest proporcjonalne do liczby użytych procesów. Istnieje zależność między zaoszczędzonym czasem obliczeniowym i czasem, jaki zajmuje włączenie kolejnego interpretera. Przykład korzysta z metody Monte Carlo (wybierania losowych liczb), aby oszacować wartość π[7].

```
>>> import multiprocessing
>>> import random
>>> import timeit
>>>
>>> def calculate_pi(iterations):
...     x = (random.random() for i in range(iterations))
...     y = (random.random() for i in range(iterations))
...     r_squared = [xi**2 + yi**2 for xi, yi in zip(x, y)]
...     percent_coverage = sum([r <= 1 for r in r_squared]) / len(r_squared)
...     return 4 * percent_coverage
...
>>>
>>> def run_pool(processes, total_iterations):
...     with multiprocessing.Pool(processes) as pool: ❶
...         # Podziel wszystkie iteracje pomiędzy procesy.
...         iterations = [total_iterations // processes] * processes ❷
...         result = pool.map(calculate_pi, iterations) ❸
...     print( "%0.4f" % (sum(result) / processes), end=', ')
...
>>>
>>> ten_million = 10000000 ❹
>>> timeit.timeit(lambda: run_pool(1, ten_million), number=10)
3.141, 3.142, 3.142, 3.141, 3.141, 3.142, 3.141, 3.141, 3.142, 3.142,
134.48382110201055 ❺
>>> ❻
>>> timeit.timeit(lambda: run_pool(10, ten_million), number=10)
3.142, 3.142, 3.142, 3.142, 3.142, 3.142, 3.141, 3.142, 3.142, 3.141,
74.38514468498761 ❼
```

❶ Użycie `multiprocessing.Pool` wewnątrz menedżera kontekstu wymusza to, że `pool` powinien być użyty tylko przez proces, który go tworzy.

❷ Pełna liczba iteracji będzie zawsze taka sama; będą po prostu podzielone na różną liczbę procesów.

❸ `pool.map()` tworzy wiele procesów — jeden na każdy element z listy `iterations`, aż do maksymalnej liczby ustalonej podczas inicjalizacji puli (w `multiprocessing.Pool(processes)`).

❹ Jest tylko jeden proces dla pierwszej próby `timeit`.

❺ 10 powtórzeń jednego procesu z 10 milionami iteracji zajęło 134 sekundy.

❻ Druga próba `timeit` obejmuje 10 procesów.

❼ 10 powtórzeń 10 procesów z milionami iteracji zajęło 74 sekundy.

[7] Pełne wyprowadzenie metody znajdziesz na stronie *http://bit.ly/monte-carlo-pi*. W skrócie — rzucasz lotkami w kwadrat o wymiarach 2×2, który zawiera w sobie okrąg o promieniu 1. Jeśli lotka trafia z równym prawdopodobieństwem w dowolne miejsce na planszy, odsetek lotek, które znajdują się w okręgu, wynosi π/4. Co oznacza, że czterokrotność tego odsetka jest równa π.

Celem tego było pokazanie, że istnieją koszty tworzenia wielu procesów, ale narzędzia do uruchamiania ich w Pythonie są solidne i dopracowane. Aby dowiedzieć się więcej, poczytaj dokumentację modułu multiprocessing w bibliotece standardowej Pythona (*https://docs.python.org/3.5/library/multiprocessing.html*). Przeczytaj także post o obchodzeniu GIL na blogu Jeffa Knuppa (*http://bit.ly/pythons-hardest-problems*), ponieważ zawiera on kilka akapitów dotyczących modułu multiprocessing.

Subprocess

Biblioteka subprocess (*https://docs.python.org/3/library/subprocess.html*) została wprowadzona do biblioteki standardowej w Pythonie 2.4 i zdefiniowana w PEP 324 (*https://www.python.org/dev/peps/pep-0324*). Uruchamia wywołania systemowe (np. `unzip` lub `curl`) tak, jakby były one wykonane z wiersza poleceń (domyślnie bez wywołania powłoki systemowej — *http://bit.ly/subprocess-security*), a programista decyduje, co zrobić z wejściem i wyjściem subprocess. Zalecamy użytkownikom Pythona 2, aby pobrali zaktualizowaną wersję z naprawionymi błędami w pakiecie subprocess32 (*https://pypi.python.org/pypi/subprocess32/*). Zainstaluj go narzędziem `pip`:

```
$ pip install subprocess32
```

Świetny samouczek subprocess znajdziesz na blogu Python Module of the Week (*https://pymotw.com/2/subprocess/*).

PyPy

PyPy (*http://pypy.org*) jest czysto pythonową implementacją Pythona. Jest szybka, a podczas jej działania nie musisz nic robić ze swoim kodem, który po prostu jest szybciej obsługiwany. Powinieneś wypróbować tę możliwość w pierwszej kolejności.

Nie możesz jej pobrać za pomocą polecenia `pip`, ponieważ jest w zasadzie kolejną interpretacją Pythona. Wyszukaj swojej wersji Pythona i systemu operacyjnego na stronie pobierania PyPy (*http://pypy.org/download.html*).

Oto nieco zmodyfikowana wersja testu CPU opracowanego przez Davida Beazleya (*http://www.dabeaz.com/GIL/gilvis/measure2.py*), z dodaną pętlą dla wielokrotnych testów. Na jej przykładzie możesz zobaczyć różnicę pomiędzy PyPy i CPythonem. Najpierw uruchomienie za pomocą CPythona:

```
$ # CPython
$ ./python -V
Python 2.7.1
$
$ ./python measure2.py
1.06774401665
1.45412397385
1.51485204697
1.54693889618
1.60109114647
```

A tutaj ten sam skrypt — jedyną różnicą jest interpreter Pythona, w tym przypadku włączony za pomocą PyPy:

```
$ # PyPy
$ ./pypy -V
Python 2.7.1 (7773f8fc4223, Nov 18 2011, 18:47:10)
[PyPy 1.7.0 with GCC 4.4.3]
```

```
$
$ ./pypy measure2.py
0.0683999061584
0.0483210086823
0.0388588905334
0.0440690517426
0.0695300102234
```

Jak widać, tylko dzięki pobraniu PyPy średni czas zmniejszył się z ok. 1,5 sekundy do ok. 0,05 sekundy — to około 30 razy szybciej. Czasami Twój kod nie przyspieszy nawet dwukrotnie, innym razem zyskasz naprawdę duże przyspieszenie — co ważne, bez wysiłku, pomijając pobranie interpretera PyPy. Jeśli chcesz, aby Twoja biblioteka C była kompatybilna z PyPy, stosuj się do zalecenia PyPy (*http://pypy.org/compat.html*) i korzystaj w standardowej bibliotece z CFFI zamiast ctypes.

Cython

Niestety, PyPy nie działa ze wszystkimi bibliotekami korzystającymi z rozszerzeń C. W takich przypadkach Cython (*http://cython.org/*) implementuje zbiór języków Pythona, który pozwala Ci na pisanie modułów C i C++ dla Pythona. Cython zezwala też na wywołanie funkcji ze skompilowanych bibliotek C i zapewnia kontekst, `nogil`, który umożliwia zwolnienie GIL (*http://tinyurl.com/cython-nogil*) w obrębie fragmentu kodu, o ile kod ten nie wpływa w żaden sposób na obiekty Pythona. Użycie Cythona pozwala Ci korzystać z silnego typowania[8] zmiennych i operacji Pythona.

Oto przykład silnego typowania w Cythonie:

```
#-*-coding:utf-8-*-
def primes(int kmax):
"""Obliczanie liczb pierwszych z dodatkowymi słowami kluczowymi Cythona"""

    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k= 0
    n= 2
    while k < kmax:
        i= 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Ta implementacja algorytmu do wyszukiwania liczb pierwszych ma dodatkowe słowa kluczowe w porównaniu z następną implementacją w czystym Pythonie:

```
#-*-coding:utf-8-*-
def primes(kmax):
"""Obliczanie liczb pierwszych ze standardową składnią Pythona"""
```

[8] Język może być jednocześnie silnie i dynamicznie typowany, jak opisano w dyskusji na Stack Overflow: *http://stackoverflow.com/questions/11328920*.

```
    p= range(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k= 0
    n= 2
    while k < kmax:
        i= 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Zauważ, że w wersji Cythona deklarujesz liczby całkowite i tablice liczb całkowitych, aby zostały skompilowane do typów C, wciąż tworząc przy tym listę Pythona:

```
#-*-coding:utf-8-*-
# Wersja Cythona

def primes(int kmax): ❶
    """Obliczanie liczb pierwszych z dodatkowymi słowami kluczowymi Cythona"""
    cdef int n, k, i ❷
    cdef int p[1000] ❸
    result = []
```

❶ Typ jest zadeklarowany jako liczba całkowita.

❷ Nadchodzące zmienne `n`, `k` oraz `i` są zadeklarowane jako liczby całkowite;

❸ A tutaj zmieniliśmy lokalizację 1000-elementowej tablicy liczb całkowitych do `p`.

Jaka jest różnica? W wersji Cythona możesz zobaczyć deklarację zmiennych typów i tablic liczb całkowitych w podobny sposób jak w standardowym C. Na przykład dodatkowa deklaracja typu (liczby całkowitej) w `cdef int n, k, i` pozwala kompilatorowi Cythona na generowanie bardziej wydajnego kodu C od tego, który zostałby wygenerowany bez podpowiedzi typu. Z powodu braku kompatybilności składni ze zwykłym Pythonem kod nie jest zapisany w plikach **.py* — zamiast tego kod Cythona jest zapisywany w plikach **.pyx*.

Jaka jest różnica w czasie? Sprawdźmy to!

```
#-*-coding:utf-8-*-
import time
# aktywuj kompilator pyx
import pyximport ❶
pyximport.install() ❷
# liczby pierwsze zaimplementowane w Cythonie
import primesCy
# liczby pierwsze zaimplementowane w Pythonie
import primes

print("Cython:")
t1 = time.time()
print primesCy.primes(500)
t2 = time.time()
print("Czas Cythona: %s" %(t2-t1))
print("")
```

```
print("Python")
t1 = time.time() ❸
print(primes.primes(500))
t2 = time.time()
print("Czas Pythona: {}".format(t2-t1))
```

❶ moduł *pyximport* pozwala Ci na importowanie plików **.pyx* (np. *primesCy.pyx*) ze skompilowaną przez Cythona wersją funkcji `primes`.

❷ Komenda `pyximport.install()` zezwala interpreterowi Pythona na włączanie kompilatora Cythona bezpośrednio do generowania kodu C, który jest automatycznie kompilowany do biblioteki C **.so*. Cython ma wtedy możliwość łatwego i wydajnego importowania takiej biblioteki dla Twojego kodu Pythona.

❸ Za pomocą funkcji `time.time()` możesz porównać czas pomiędzy tymi dwoma wywołaniami do znalezienia 500 liczb pierwszych. W standardowym notatniku (dual-core AMD E-450 1,6 GHz) zmierzone wartości to:

```
Czas Cythona: 0.0054 seconds
Czas Pythona: 0.0566 seconds
```

A oto wyjście z wbudowanej maszyny ARM BeagleBone (*http://beagleboard.org/Products/BeagleBone*):

```
Czas Cythona: 0.0196 seconds
Czas Pythona: 0.3302 seconds
```

Numba

Numba (*http://numba.pydata.org*) to przystosowany do współpracy z NumPy kompilator Pythona (specjalizujący się w kompilacji *just-in-time* [JIT]), który kompiluje adnotowany kod Pythona (i NumPy) do LLVM (Low-Level Virtual Machine — *http://llvm.org/*) przez specjalne dekoratory. Pokrótce Numba korzysta z LLVM do kompilacji Pythona do kodu maszynowego, który może być naturalnie wykonywany w czasie wykonywania.

Jeśli korzystasz z Anacondy, zainstaluj Numbę za pomocą polecenia `conda install Numba`; jeśli nie, zainstaluj ją ręcznie. Przed instalacją Numby musisz mieć zainstalowane NumPy i LLVM.

Sprawdź, jakiej wersji LLVM potrzebujesz (możesz to znaleźć na stronie PyPI dla llvmlite — *https://pypi.python.org/pypi/llvmlite*), i pobierz wersję z lokalizacji, która pasuje do Twojego systemu:

- LLVM dla Windowsa: *http://llvm.org/builds/*.
- LLVM dla Debiana (Ubuntu): *http://llvm.org/apt/*.
- LLVM dla Fedory: *https://apps.fedoraproject.org/packages/llvm*.
- Aby zapoznać się z dyskusją dotyczącą budowania ze źródła na innym systemie Uniksowym, pobierz artykuł „Building the Clang + LLVM compilers” (*http://ftp.math.utah.edu/pub/llvm/*).
- Na macOS użyj `brew install homebrew/versions/llvm37` (lub innej aktualnej wersji).

Kiedy już zainstalujesz LLVM i Numpy, zainstaluj Numbę za pomocą narzędzia `pip`. Możesz pomóc instalatorowi w znalezieniu pliku *llvm-config* przez zapewnienie dodatkowej zmiennej środowiskowej `LLVM_CONFIG` z odpowiednią ścieżką:

```
$ LLVM_CONFIG=/path/to/llvm-config-3.7 pip install numba
```

Aby następnie z niej skorzystać, po prostu dekoruj swoje funkcje:

```
from numba import jit, int32

@jit ❶
def f(x):
    return x + 3
@jit(int32(int32, int32)) ❷
def g(x, y):
    return x + y
```

❶ Bez argumentów dekorator `@jit` wykonuje leniwą kompilację — decyduje sam, czy i jak optymalizować funkcję.

❷ Dla *gorliwej kompilacji* sprecyzuj typy. Funkcje zostaną skompilowane z zadaną specjalizacją, a żadna inna nie będzie dozwolona — zwracana wartość i dwa argumenty będą miały typ `numba.int32`.

Istnieje flaga `nogil`, która pozwala ignorować GIL, oraz moduł `numba.pycc`, który może zostać użyty do kompilacji kodu przed czasem. Aby dowiedzieć się więcej, zobacz instrukcję użytkowania Numby (*http://numba.pydata.org/numba-doc/latest/user*).

Biblioteki GPU

Numba może opcjonalnie zostać zbudowana ze zdolnością do działania na *procesorze graficznym* (GPU), czipie zoptymalizowanym do szybkich, równoległych obliczeń, wykorzystywanych w nowoczesnych grach. W tym celu będziesz musiał posiadać procesor graficzny firmy NVIDIA z zainstalowanym zestawem narzędzi CUDA (*https://developer.nvidia.com/cuda-downloads*). Następnie stosuj się do dokumentacji korzystania z CUDA JIT Numby (*http://numba.pydata.org/numba-doc/0.13/CUDAJit.html*) w połączeniu z GPU.

Inną popularną biblioteką ze zdolnościami wykorzystania GPU jest biblioteka TensorFlow (*https://www.tensorflow.org*), wydana przez Google na licencji Apache 2.0. Zapewnia ona tensory (wielowymiarowe macierze) i sposób na łączenie operacji na tensorach w celu wykonywania szybkich obliczeń na macierzach. Aby ją zainstalować, stosuj się do instrukcji ze strony *https://www.tensorflow.org/install/*.

Theano (*http://deeplearning.net/software/theano/*) to dobra alternatywa dla TensorFlow, jeśli pracujesz na macOS (który nie wspiera operacji na GPU). Theano była biblioteką do matematyki na macierzach w procesorach graficznych, zanim Google wydało TensorFlow, i wciąż jest aktywnie rozwijana. Ma stronę poświęconą wykorzystaniu GPU (*http://deeplearning.net/software/theano/tutorial/using_gpu.html*). Theano działa na Windowsie, systemach linuksowych i macOS. Jest dostępna przez polecenie `pip`:

```
$ pip install Theano
```

Dla niskopoziomowych operacji na GPU możesz wypróbować PyCUDA (*https://developer.nvidia.com/pycuda*).

Natomiast osoby nieposiadające procesora graficznego firmy NVIDIA mogą wykorzystać PyOpenCL (*https://pypi.python.org/pypi/pyopencl*), opakowanie biblioteki OpenCL Intela (*https://software.intel.com/en-us/intel-opencl*), które jest kompatybilne z wieloma różnymi ustawieniami sprzętowymi (*https://software.intel.com/en-us/articles/opencl-drivers*).

Wchodzenie w interakcję z bibliotekami C, C++, FORTRANA

Biblioteki opisane w kolejnych sekcjach znacznie różnią się od siebie: zarówno CFFI i ctypes to biblioteki Pythona, F2PY jest dla FORTRANA, SWIG udostępnia obiekty C dla wielu języków (nie tylko dla Pythona), a Boost.Python jest biblioteką C++, która może odsłaniać obiekty C++ dla Pythona i odwrotnie. Tabela 8.2 opisuje wszystkie te biblioteki nieco bardziej szczegółowo.

Tabela 8.2. Interfejsy C i C++

Biblioteka	Licencja	Powody użycia
CFFI	Licencja MIT	• Zapewnia najlepszą kompatybilność z PyPy. • Umożliwia pisanie kodu C z wnętrza Pythona, który może być skompilowany do tworzenia biblioteki C z wiązaniami Pythona.
ctypes	Licencja Python Software Foundation	• Znajduje się w bibliotece standardowej Pythona. • Pozwala na opakowanie istniejącej DLL lub dzielonego obiektu, którego nie napisałeś lub nad którym nie masz kontroli. • Ustępuje tylko CFFI pod względem kompatybilności z Pythonem.
F2PY	Licencja BSD	• Pozwala na korzystanie z biblioteki FORTRANA. • F2PY jest częścią NumPY, więc powinieneś korzystać z NumPy.
SWIG	GPL (wyjście nie jest ograniczone)	• Zapewnia sposób na automatyczne generowanie bibliotek w wielu językach z wykorzystaniem specjalnego formatu plików, który nie jest ani z C, ani z Pythona.
Boost.Python	Licencja Boost Software	• To nie narzędzie konsolowe, ale biblioteka C++, która może być włączona do kodu C++ i wykorzystana do określenia, które obiekty eksponować w Pythonie.

C Foreign Function Interface

Pakiet CFFI (*https://cffi.readthedocs.org/en/latest/*) zapewnia prosty mechanizm komunikacji z C z CPythona oraz PyPy. CFFI jest rekomendowany przez PyPy (*http://doc.pypy.org/en/latest/extending.html*) z powodu najlepszej kompatybilności pomiędzy CPythonem i PyPy. Obsługuje dwa tryby: kompatybilny *interfejs binarny aplikacji* (ABI — zobacz poniższy przykład kodu), który pozwala na dynamiczne ładowanie i wywoływanie funkcji z modułu wykonywalnego (właściwie eksponując tę samą funkcjonalność co LoadLibrary lub dlopen) i tryb API, który pozwala Ci na budowę modułów rozszerzeń C[9].

Zainstaluj pakiet za pomocą narzędzia `pip`:

```
$ pip install cffi
```

[9] Podczas pisania rozszerzeń C powinieneś zwrócić szczególną uwagę na to, czy zarejestrowałeś swoje wątki z interpreterem (*http://docs.python.org/c-api/init.html - threads*).

Oto przykład z interakcją ABI:

```
#-*-coding:utf-8-*-
from cffi import FFI
    ffi = FFI()
    ffi.cdef("size_t strlen(const char*);") ❶
    clib = ffi.dlopen(None) ❷
    length = clib.strlen("String do oceny.") ❸
    # prints: 23
    print("{}".format(length))
```

❶ Ten łańcuch mógłby wziąć się z deklaracji funkcji w pliku nagłówkowym C.

❷ Otwórz dzieloną bibliotekę (**.DLL* lub **.so*).

❸ Teraz możemy traktować `clib` tak, jakby był modułem Pythona, i wykonywać funkcje, które zdefiniowaliśmy za pomocą notacji z kropką.

ctypes

ctypes (*https://docs.python.org/3/library/ctypes.html*) to tak naprawdę biblioteka do komunikacji z C i C++ z poziomu CPythona, która znajduje się w bibliotece standardowej. Zapewnia pełny dostęp do naturalnego interfejsu C na większości systemów operacyjnych (np. kernel32 w Windowsie lub libc w *nix), a do tego obsługę ładowania i komunikacji z dynamicznymi bibliotekami — dzielonymi obiektami (*.so*) lub DLL — w czasie wykonania. Zawiera też całą masę typów do interakcji z API systemowym i pozwala na łatwe zdefiniowanie własnych złożonych typów, takich jak struktury czy unie, oraz modyfikowanie takich właściwości, jak padding czy wyrównanie. Może być trudna w użyciu (ponieważ musisz pisać tak dużo dodatkowych znaków), ale w połączeniu z modułem `struct` (*https://docs.python.org/3.5/library/struct.html*) z biblioteki standardowej daje pełną kontrolę nad tym, w jaki sposób typy danych są tłumaczone na coś użytecznego dla metod w czystym C lub C++.

Na przykład struktura C zdefiniowana w pliku nazwanym *my_struct.h*

```
struct my_struct {
    int a;
    int b;
};
```

może być zaimplementowana w poniższy sposób w pliku *my_struct.py*:

```
    import ctypes
    class my_struct(ctypes.Structure):
        _fields_ = [("a", c_int),
                    ("b", c_int)]
```

F2PY

Generator interfejsu FORTRAN-Python (F2PY — *http://docs.scipy.org/doc/numpy/f2py/*) jest częścią NumPy, więc pobierz NumPy za pomocą narzędzia `pip`:

```
$ pip install numpy
```

Zapewnia on wszechstronną funkcję konsolową `f2py`, która może być użyta na trzy różne sposoby, wszystkie udokumentowane w przewodniku rozpoczynania pracy z F2PY (*http://docs.scipy.org/doc/numpy/f2py/getting-started.html*). Jeśli masz kontrolę nad kodem źródłowym, możesz dodać

specjalne komentarze z instrukcjami dla F2PY, które rozjaśniają zamiary każdego argumentu (precyzują, które elementy są zwracane, a które przyjmowane), a następnie uruchomić F2PY:

```
$ f2py -c fortran_code.f -m python_module_name
```

Kiedy nie masz dostępu, F2PY może wygenerować pośredni plik z rozszerzeniem **.pyf*, który możesz modyfikować, a następnie dojść do tego samego wyniku. Można to zrobić w trzech krokach:

```
$ f2py fortran_code.f -m python_module_name -h interface_file.pyf ❶
$ vim interface_file.pyf ❷
$ f2py -c interface_file.pyf fortran_code.f ❸
```

❶ Automatyczne generowanie pośredniego pliku, który definiuje interfejs pomiędzy sygnaturami funkcji FORTRAN-a i sygnaturami Pythona.

❷ Edytuj plik, aby poprawnie oznaczał zmienne wejściowe i wyjściowe.

❸ Teraz skompiluj kod i utwórz moduły rozszerzeń.

SWIG

SWIG (*Simplified Wrapper Interface Generator* — dosł. „uproszczone opakowanie generatora interfejsu"; *http://www.swig.org*) obsługuje dużą liczbę języków skryptowych, włączając Pythona. Jest popularnym, często stosowanym narzędziem konsolowym, które generuje wiązania dla języków interpretowanych z plików nagłówkowych C i C++. Aby z niego korzystać, najpierw użyj SWIG do automatycznego wygenerowania pośredniego pliku z nagłówka — z sufiksem **.i*. Następnie zmodyfikuj ten plik, aby odzwierciedlał rzeczywisty interfejs, którego potrzebujesz. Wszystko to jest opisane krok po kroku w samouczku SWIG (*http://www.swig.org/tutorial.html*).

Chociaż SWIG ma pewne ograniczenia (obecnie zdaje się mieć problemy z małym podzbiorem nowszych funkcji C++, a zmuszenie do działania kodu pełnego szablonów może być dość pracochłonne), zapewnia wielką moc i z symbolicznym wysiłkiem eksponuje wiele funkcji dla Pythona. Dodatkowo możesz łatwo rozszerzać wiązania, które SWIG tworzy (w pliku interfejsu), aby przeciążać operatory i wbudowane metody, oraz efektywnie wykorzystywać wyjątki C++, tak aby Python mógł je przechwytywać.

Oto przykład, który pokazuje, jak przeciążyć `__repr__`. Ten fragment mógłby znaleźć się w pliku nazwanym *MyClass.h*:

```
#include <string>
class MyClass {
private:
    std::string name;
public:
    std::string getName();
};
```

A oto *myclass.i*:

```
%include "string.i"

%module myclass
%{
#include <string>
#include "MyClass.h"
%}
```

```
%extend MyClass {
    std::string __repr__()
    {
        return $self->getName();
    }
}
%include "MyClass.h"
```

W repozytorium SWIG na GitHubie (*https://github.com/swig/swig/tree/master/Examples/python*) znajduje się więcej przykładów z Pythona. Zainstaluj SWIG, korzystając ze swojego menedżera pakietów, jeśli tam jest (`apt-get install swig`, `yum install swig.i386` lub `brew install swig`), lub pobierz go ze strony *http://www.swig.org/survey.html*, a następnie stosuj się do instrukcji instalacji (*http://www.swig.org/Doc3.0/Preface.html#Preface_installation*) dla swojego systemu operacyjnego. Jeśli nie masz na swoim komputerze z macOS biblioteki Perl Compatible Regular Expressions (PCRE), do jej instalacji wykorzystaj Homebrew:

```
$ brew install pcre
```

Boost.Python

Boost.Python (*http://www.boost.org/doc/libs/1_60_0/libs/python/doc/*) wymaga odrobinę więcej pracy manualnej do wyeksponowania funkcjonalności obiektów C++, ale potrafi zapewnić takie same funkcje jak SWIG, a nawet więcej — np. opakowania w celu osiągnięcia dostępu do obiektów Pythona, jak PyObjects w C++, a także narzędzia do eksponowania obiektów C++ dla Pythona. W porównaniu ze SWIG Boost.Python jest biblioteką, a nie narzędziem konsolowym, a do tego nie ma potrzeby tworzenia pośredniego pliku z innym formatowaniem — wszystko jest napisane bezpośrednio w C++. Jeśli chcesz z niego korzystać, Boost.Python ma obszerny, szczegółowy samouczek (*http://bit.ly/boost-python-tutorial*).

O'REILLY®

Sprawdź, jak wykorzystać siłę zwinności!

Python to potężny, cechujący się prostotą i elastycznością język ułatwiający wydajne tworzenie czytelnego kodu. Nadaje się do różnych zastosowań, włączając w to wysoce specjalistyczne zadania, takie jak analiza danych. Python cieszy się niezwykłym wsparciem społeczności, dzięki czemu powstają nowe, ciekawe narzędzia dla programistów. To wszystko sprawia, że dla twórców oprogramowania biegłe posługiwanie się Pythonem staje się niezbędną umiejętnością.

Niniejsza książka jest przeznaczona dla średnio zaawansowanych programistów. Zawarto tu zbiór najlepszych praktyk i opis ulubionych narzędzi entuzjastów Pythona. Przedstawiono doskonałe biblioteki do aplikacji konsolowych, graficznych interfejsów i aplikacji internetowych oraz do analizy danych, obróbki zdjęć i dźwięku, a także biblioteki sieciowe do akcji asynchronicznych, serializacji i kryptografii. Znalazły się tu liczne przykłady fragmentów kodu, opisano również dobre praktyki pakietowania i dystrybucji kodu. Książka ta stanowi cenne źródło informacji dla każdego, kto chce nabrać biegłości w posługiwaniu się językiem Python.

Najważniejsze zagadnienia przedstawione w książce:

- edytory kodu, środowiska programistyczne i interpretery Pythona
- styl kodu, konwencje i idiomy oraz struktura aplikacji
- techniki testowania aplikacji
- operacje na danych, data mining i zastosowania naukowe
- praca na bazach danych

Kenneth Reitz — jest znanym projektantem oprogramowania, członkiem Python Software Foundation. Chętnie propaguje ideę oprogramowania *open source* — brał udział w tworzeniu wielu takich projektów, na przykład Requests: HTTP for Humans.

Tanya Schlusser — biegle posługuje się kilkoma językami programowania. Zajmuje się również głęboką analizą danych i ich wykorzystywaniem w podejmowaniu decyzji strategicznych. Jest członkinią Chicago Python User's Group i Chicago's PyLadies. Pracuje jako niezależna konsultantka — szkoli studentów i firmowe zespoły analityków danych.

ISBN 978-83-283-3732-9
9 788328 337329
Cena: 54,90 zł