O'REILLY®

TDD w praktyce

Niezawodny kod w języku Python

TWÓRZ NIEZAWODNE APLIKACJE W JĘZYKU PYTHON!

Helion

Harry J.W. Percival

Tytuł oryginału: Test-Driven Development with Python

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-1380-4

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/tddwpr.zip

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

Pochwały dla książki *TDD w praktyce*

„W tej książce Harry zabiera nas w podróż mającą na celu odkrycie Pythona i testowania. To doskonała książka, przyjemna w lekturze i wypełniona przydatnymi informacjami. Gorąco polecam ją każdemu zainteresowanemu tematem testowania w Pythonie, poznaniem frameworka Django lub użycia narzędzia Selenium. Testowanie ma niezwykle istotne znaczenie w pracy programisty, to niewątpliwie trudny obszar, pełen kompromisów. Harry spisał się doskonale — przyciągnął naszą uwagę, wyjaśniając praktyki stosowane podczas testowania".

— *Michael Foord*
Programista Pythona i jednocześnie osoba odpowiedzialna za rozwój modułu unittest

„Ta książka to znacznie więcej niż tylko wprowadzenie do programowania sterowanego testami w Pythonie. To jest pełny kurs przedstawiający najlepsze praktyki od początku do końca na przykładzie nowoczesnego programowania aplikacji sieciowej w Pythonie. Każdy programista sieciowy powinien zapoznać się z tą książką".

— *Kenneth Reitz*
Członek Python Software Foundation

„Żałujemy, że nie mieliśmy do dyspozycji książki Harry'ego, gdy poznawaliśmy Django. Zachowuje ona odpowiednie tempo, a przy tym jest nieco wymagająca. Stanowi doskonałe wprowadzenie do frameworka Django oraz różnych praktyk z zakresu testowania. Tę książkę warto kupić choćby dla samego materiału poświęconego Selenium, choć oczywiście znajdziesz w niej dużo więcej!"

— *Daniel i Audrey Greenfieldowie*
autorzy książki *Two Scoops of Django* (Two Scoops Press)

Spis treści

Wprowadzenie

Niniejszą książkę traktuję jako swego rodzaju próbę podzielenia się z czytelnikami moimi doświadczeniami, jakie zdobyłem w trakcie podróży od „hakera" do „inżyniera oprogramowania". Znajdziesz tutaj informacje przede wszystkim z zakresu testowania, ale jak się wkrótce przekonasz, także z wielu innych dziedzin.

Na początek dziękuję, że sięgnąłeś po tę książkę.

Jeśli kupiłeś jej egzemplarz, tym bardziej jestem wdzięczny. Natomiast jeśli czytasz bezpłatną wersję opublikowaną w internecie, to *nadal* jestem wdzięczny, że zdecydowałeś się poświęcić swój czas na lekturę. Kto wie, być może gdy dotrzesz do końca książki, uznasz ją za wartą zakupu dla siebie lub przyjaciela.

Jeżeli masz jakiekolwiek komentarze, pytania lub sugestie, podziel się nimi ze mną. Jestem dostępny bezpośrednio poprzez e-mail *obeythetestinggoat@gmail.com* lub w serwisie Twitter (*@hjwp*)[1]. Znajdziesz mnie również na *forum tematycznym książki*[2]. Zajrzyj także na moją *witrynę i bloga*[3].

Mam nadzieję, że lektura książki sprawiła Ci tyle radości, ile mnie jej napisanie.

Dlaczego napisałem książkę o programowaniu sterowanym testami?

Już słyszę, jak pytasz: „Kim jesteś, dlaczego napisałeś tę książkę i dlaczego miałbym ją przeczytać?".

Nadal jestem na początku mojej kariery programisty. Można spotkać się ze stwierdzeniem, że w każdej dyscyplinie zaczyna się od etapu ucznia, następnie przechodzi do czeladnika i na koniec czasami staje się mistrzem. O sobie mógłbym powiedzieć, że jestem (co najwyżej) czeladnikiem programowania. Na wczesnym etapie kariery miałem dużo szczęścia i zetknąłem się z wieloma fanatykami stosowania technik TDD w programowaniu, co oczywiście wywarło ogromny wpływ na sposób, w jaki programuję. Zapragnąłem więc podzielić się swoimi doświadczeniami z innymi. Mógłbyś powiedzieć, że mój entuzjazm jest wynikiem ostatnich zmian i nadal zachowuję świeże wspomnienia z niedawnej nauki, więc łatwo mogę wczuć się w położenie początkujących.

[1] *https://twitter.com/hjwp*

[2] *https://groups.google.com/forum/#!forum/obey-the-testing-goat-book*

[3] *http://www.obeythetestinggoat.com/*

Kiedy dopiero poznawałem Pythona (na podstawie doskonałej książki Marka Pilgrima zatytułowanej *Dive Into Python*), natknąłem się na koncepcję stosowania technik TDD i stwierdziłem: „tak, zdecydowanie widzę w tym sens". Prawdopodobnie miałeś podobne odczucie, gdy po raz pierwszy zetknąłeś się z TDD. Techniki TDD wyglądają na całkiem rozsądne podejście i naprawdę stanowią dobry nawyk wart stosowania, podobnie jak na przykład regularne szczotkowanie zębów.

Następnie pojawił się mój pierwszy duży projekt i możesz zgadnąć, co się stało — miałem klienta, napięte terminy, dużo pracy do wykonania i wszelkie dobre intencje dotyczące technik TDD szybko odeszły w kąt.

Wszystko było w porządku, a ja czułem się dobrze.

Przynajmniej na początku.

Na początku wiedziałem, że tak naprawdę nie potrzebuję technik TDD, ponieważ pracuję nad małą witryną internetową. Bardzo łatwo mogłem ręcznie sprawdzić działanie poszczególnych składników witryny. Kliknij *tutaj* łącze, wybierz *tam* element z rozwijanej listy, a *wtedy* powinno wydarzyć się to i to. Całkiem łatwe. Tworzenie testów do sprawdzania wymienionych rzeczy trwałoby *wieki* lub jeszcze dłużej. Poza tym na podstawie pełnych trzech tygodni tworzenia kodu wydawało mi się, że stałem się całkiem dobrym programistą i mogę sobie poradzić z projektem. Łatwizna.

Następnie nadszedł strach przed boginią Złożoności, która wkrótce się pojawiła i pokazała mi braki w moim doświadczeniu.

Projekt rozrastał się i pewne komponenty systemu zaczęły zależeć od innych. Robiłem wszystko, co w mojej mocy, aby kierować się dobrymi zasadami, takimi jak DRY (nie powtarzaj się), ale to zaprowadziło mnie na całkiem niebezpieczne tereny. Wkrótce doszło wielokrotne dziedziczenie, hierarchie klas na ośmiu poziomach głębokości i polecenia `eval`.

Zacząłem być przerażony na myśl o wprowadzeniu jakiejkolwiek zmiany w kodzie. Nie byłem już pewien, co jest zależne od czego i co może się stać, gdy wprowadzę zmianę *tego* fragmentu kodu — jejku, myślałem, że ten kod dziedziczy po tamtym. Nie, tutaj został nadpisany. O nie, jego działanie zależy od zmiennej klasy. Dobrze, jeżeli nadpiszę nadpisany wcześniej fragment, to wszystko będzie dobrze. Tylko to sprawdzę. Jednak operacja sprawdzania okazała się dużo trudniejsza, niż sądziłem. Budowana witryna internetowa zawiera teraz wiele sekcji i ręczne ich sprawdzanie zaczęło być po prostu niepraktyczne. Lepiej pozostawić wszystko, jak jest, zapomnieć o refaktoryzacji i skoncentrować się na tworzeniu.

Bardzo szybko doprowadziłem do powstania okropnego kodu, a tworzenie nowego stało się prawdziwym utrapieniem.

Niedługo potem miałem szczęście i otrzymałem pracę w firmie Resolver Systemem (obecnie *PythonAnywhere*[4]), w której stosowanie programowania ekstremalnego (ang. *extreme programming* — XP) było normą. Pracownicy firmy przyuczyli mnie do rygorystycznego stosowania technik TDD.

[4] *https://www.pythonanywhere.com/*

Wprawdzie wcześniej zdobyte doświadczenie na pewno otworzyło mój umysł na potencjalne korzyści wynikające z automatyzowanego testowania, ale nadal na każdym etapie pojawiały się pewne wątpliwości. „Ogólnie rzecz biorąc, testowanie może być dobrym pomysłem, ale *naprawdę*?" Wszystkie te testy? Niektóre z nich wydawały się zupełną stratą czasu… Co takiego? Testy funkcjonalne *oraz* testy jednostkowe? Dajcie spokój, to już przesada! I na dodatek ten cykl TDD: testowanie, minimalna zmiana w kodzie i znów testowanie. To już naprawdę głupota! Nie musimy wykonywać tych dziecinnych kroków. Dajcie spokój, przecież widać, jak wygląda prawidłowa odpowiedź. Dlaczego nie możemy pominąć tych testów do końca?

Możesz mi uwierzyć: przewidywałem każdą regułę, sugerowałem każdy skrót, domagałem się uzasadnienia dla każdego aspektu TDD z pozoru wydającego się bezsensownym i poddałem się, widząc sens w tym wszystkim. Niezliczoną ilość razy powtarzałem sobie „dzięki wam, testy", gdy test funkcjonalny wskazał regresję, której nigdy nie przewidywałem, lub test jednostkowy uchronił mnie przed popełnieniem naprawdę głupiego błędu logicznego. Pod względem psychologicznym tego rodzaju programowanie jest mniej stresującym procesem. Efektem jest kod, z którym praca będzie przyjemnością.

Dlatego też pozwól mi o tym *wszystkim* Ci opowiedzieć.

Cel książki

Moim podstawowym celem jest przekazanie metodologii — sposobu programowania sieciowego, który prowadzi do powstania lepszych aplikacji sieciowych i jednocześnie zapewnia więcej radości ich programistom. Nie ma zbyt dużego sensu w pisaniu książki przedstawiającej materiał, który można znaleźć, korzystając z ulubionej wyszukiwarki internetowej. Dlatego też to nie jest przewodnik omawiający składnię Pythona lub programowanie sieciowe *jako takie*. Zamiast tego mam nadzieję nauczyć Cię, jak wykorzystać techniki TDD do osiągnięcia wymarzonego celu: *przejrzystego kodu, który po prostu działa*.

Mając to na uwadze, nieustannie odwołuję się do rzeczywistego, praktycznego projektu i buduję aplikację sieciową zupełnie od początku, używając do tego narzędzi takich jak Django, Selenium, jQuery i Mock. Nie przyjąłem żadnych założeń dotyczących Twojej wiedzy z zakresu wymienionych narzędzi i dlatego po zakończeniu lektury będziesz miał solidne podstawy dotyczące tych narzędzi, a także opanowane stosowanie technik TDD.

Programowanie ekstremalne zawsze przeprowadzane jest w parach. Dlatego też pisząc tę książkę, wyobrażałem sobie parę z samym sobą, starałem się wytłumaczyć sposób działania narzędzi oraz odpowiadać na pytania dotyczące określonych sposobów działania kodu. Jeżeli gdziekolwiek piszę zbyt protekcjonalnie, wynika to z niewystarczającego sprytu oraz braku zachowania cierpliwości względem samego siebie. Natomiast jeśli piszę zbyt defensywnie, wynika to z faktu, że jestem irytującą osobą, która systematycznie nie zgadza się z innymi. Dlatego czasami potrzebuję wielu uzasadnień, zanim przekonam się do czegoś.

Zarys książki

Książkę podzieliłem na trzy części.

Część I (rozdziały od 1. do 6.): Podstawy
: Od razu przystępujemy do budowy prostej aplikacji sieciowej, wykorzystując przy tym techniki TDD. Pracę rozpoczynamy od utworzenia testu funkcjonalnego (za pomocą narzędzia Selenium), a następnie przechodzimy przez podstawy Django — modele, widoki i szablony — na każdym etapie rygorystycznie stosując testy jednostkowe. Ponadto dowiesz się, czym jest Testing Goat[5].

Część II (rozdziały od 7. do 14.): Programowanie sieciowe
: W tej części znajdziesz omówienie trudniejszych, choć niemożliwych do uniknięcia aspektów programowania sieciowego. Zobaczysz także, jak testy pomagają radzić sobie ze wspomnianymi aspektami: plikami statycznymi, wdrożeniem w środowisku produkcyjnym, weryfikacją danych formularza, migracjami bazy danych i budzącym lęk JavaScriptem.

Część III (rozdziały od 15. do 20.): Bardziej zaawansowane zagadnienia
: Omówiono zagadnienia takie jak imitacje, integracja opracowanego przez firmę trzecią systemu uwierzytelniania, Ajax, konfiguracja testów, podejście Outside-In i ciągła integracja (ang. *continuous integration* — CI).

A teraz kilka słów w kwestiach porządkowych…

Konwencje zastosowane w książce

W tej książce zastosowano następujące konwencje typograficzne.

Kursywa
: Wskazuje na nowe pojęcia, adresy URL i e-mail, nazwy plików, rozszerzenia plików i tak dalej.

`Czcionka o stałej szerokości`
: Użyta jest w przykładowych fragmentach kodu, a także w samym tekście w celu odwołania się do pewnych poleceń bądź innych elementów programistycznych, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, polecenia i słowa kluczowe.

`Pogrubiona czcionka o stałej szerokości`
: Użyta w celu wyeksponowania poleceń bądź innego tekstu, który powinien być wprowadzony przez czytelnika.

Czasami używam znaków:

```
[...]
```

do wskazania, że pewna zawartość została pominięta, nastąpiło skrócenie danych wyjściowych lub przejście do ważniejszych informacji w danym fragmencie.

[5] Testing Goat to nieoficjalna maskotka TDD, więcej na ten temat znajdziesz w rozdziale 1. — *przyp. tłum.*

Taka ikona oznacza wskazówkę lub sugestię.

Taka ikona oznacza ogólną uwagę.

Ta ikona oznacza ostrzeżenie.

Użycie przykładowych kodów

Przykładowe fragmenty kodów znajdziesz w serwisie GitHub (*https://github.com/hjwp/book-example/*). Kody dla poszczególnych rozdziałów są również dostępne oddzielnie, na przykład pod adresem *https://github.com/hjwp/book-example/tree/chapter_03*. Na końcu rozdziałów znajdziesz pewne sugestie dotyczące sposobów pracy z repozytorium.

Książka ta ma na celu pomóc Ci w pracy. Ogólnie rzecz biorąc, można wykorzystywać przykłady z niej w swoich programach i w dokumentacji. Nie trzeba kontaktować się z nami w celu uzyskania zezwolenia, dopóki nie powiela się znaczących ilości kodu. Na przykład pisanie programu, w którym znajdzie się kilka fragmentów kodu z tej książki, nie wymaga zezwolenia, jednak sprzedawanie lub rozpowszechnianie płyty CD-ROM zawierającej przykłady z książki wydawnictwa O'Reilly już tak. Odpowiedź na pytanie przez cytowanie tej książki lub przykładowego kodu nie wymaga zezwolenia, ale włączenie wielu przykładowych kodów z tej książki do dokumentacji produktu czytelnika już tak.

Jesteśmy wdzięczni za umieszczanie przypisów, ale nie wymagamy tego. Przypis zwykle zawiera tytuł, autora, wydawcę i ISBN. Na przykład: Harry Percival, *TDD w praktyce. Niezawodny kod w języku Python*, ISBN 978-1-449-36482-3, Helion, Gliwice 2015.

Przygotowania i założenia

W tym miejscu znajdziesz ogólny zarys przyjętych przeze mnie założeń dotyczących Twojej wiedzy, a także oprogramowania, jakie powinieneś mieć zainstalowane w komputerze.

Python 3 i programowanie

Wprawdzie niniejszą książkę napisałem, nie zapominając o początkujących, ale jeżeli dopiero zaczynasz przygodę z programowaniem, to powinieneś przynajmniej poznać podstawy języka Python. Jeśli jeszcze nie opanowałeś podstaw, zachęcam Cię do lektury samouczka dla początkujących programistów Pythona, książki przeznaczonej dla początkujących (na przykład *Dive Into Python*[1] lub *Learn Python the Hard Way*[2]) bądź też tak dla rozrywki do zapoznania się z materiałami w witrynie *Invent Your Own Computer Games*[3]. Wszystkie wymienione zasoby stanowią doskonałe wprowadzenie do programowania w Pythonie.

Jeżeli masz doświadczenie w programowaniu w innych językach niż Python, powinieneś sobie poradzić bez problemów, ponieważ Python jest niezwykle łatwy do zrozumienia.

Kod przedstawiony w książce został utworzony w języku *Python 3*. Kiedy pisałem tę książkę w latach 2013 – 2014, wersja 3 była dostępna już od dłuższego czasu. Osiągnęliśmy więc punkt, w którym to preferowana wersja Pythona. Możesz korzystać z dowolnego systemu operacyjnego: OS X, Windows lub Linux. Nieco dalej znajdziesz informacje szczegółowe przydatne dla użytkowników poszczególnych systemów operacyjnych.

Kod przedstawiony w książce został przetestowany za pomocą Pythona w wersjach 3.3 i 3.4. Jeżeli z jakiegokolwiek powodu używasz wersji 3.2, możesz zauważyć drobne różnice. Dlatego też najlepszym rozwiązaniem będzie uaktualnienie używanej wersji, o ile masz taką możliwość.

Nie zalecam podejmowania prób użycia Pythona 2, ponieważ różnice względem nowszych wersji są już znaczące. Jeżeli Twój kolejny projekt ma powstać właśnie w Pythonie 2, to materiał przedstawiony w książce nadal może być przydatny. Jednak czas poświęcony na ustalenie, czy różnice między otrzymanymi przez Ciebie danymi wyjściowymi i przedstawionymi

[1] *http://www.diveintopython.net/*

[2] *http://learnpythonthehardway.org/*

[3] *http://inventwithpython.com/*

w książce wynikają z faktu użycia Pythona 2, czy jednak są skutkiem błędu, tak naprawdę będzie czasem zmarnowanym.

Jeżeli planujesz wykorzystanie *PythonAnywhere*[4] (startup PaaS, dla którego pracuję) zamiast lokalnie zainstalowanego Pythona, wówczas przed rozpoczęciem pracy powinieneś zapoznać się z dodatkiem A.

Niezależnie od stosowanego środowiska pracy oczekuję, że masz dostęp do Pythona, wiesz, jak uruchomić go z poziomu powłoki (najczęściej za pomocą polecenia **`python3`**), a także wiesz, jak edytować i uruchamiać pliki Pythona. Jeżeli masz jakiekolwiek wątpliwości, zawsze możesz sięgnąć do wymienionych wcześniej książek i zasobów.

Jeżeli masz zainstalowanego Pythona w wersji 2 i obawiasz się, że instalacja wersji 3 może uszkodzić istniejącą, Twoje obawy są zupełnie niepotrzebne! Obie wersje Pythona mogą bez problemu funkcjonować w jednym systemie, a swoje pakiety przechowują w zupełnie różnych miejscach. Musisz się jedynie upewnić o przygotowaniu oddzielnych poleceń do uruchamiania poszczególnych wersji Pythona. Jednego przeznaczonego dla Pythona 3 (`python3`) i drugiego dla Pythona 2 (to najczęściej po prostu `python`). Podobnie podczas instalacji narzędzia `pip` dla Pythona 3 upewniamy się, że uruchamiające go polecenie (zwykle `pip3`) jest inne od polecenia przeznaczonego do uruchamiania narzędzia `pip` w Pythonie 2.

Jak działa HTML?

Przyjąłem również założenie, że masz ogólną wiedzę dotyczącą sposobu działania sieci — czym jest HTML, żądania `POST` itd. Jeżeli nie dysponujesz wspomnianymi podstawami, powinieneś poszukać odpowiednich źródeł dotyczących HTML; kilka znajdziesz w witrynie *http://www.webplatform.org/*. Jeżeli potrafisz utworzyć w komputerze stronę HTML, wyświetlić ją w przeglądarce internetowej, a także rozumiesz sposób działania formularzy sieciowych, to masz odpowiednią wiedzę.

JavaScript

W części drugiej książki pojawi się odrobina kodu JavaScript. Jeżeli nie znasz języka JavaScript, nie przejmuj się tym teraz. Gdy na późniejszym etapie poczujesz się nieco zagubiony, wskażę Ci kilka przydatnych zasobów.

Wymagane oprogramowanie

Poza językiem Python będziemy potrzebowali jeszcze kilku innych komponentów.

Przeglądarka internetowa Firefox
: Jeżeli skorzystasz z ulubionej wyszukiwarki internetowej, szybko znajdziesz instalator przeglądarki Firefox dla używanego systemu operacyjnego. Wprawdzie narzędzie Selenium współpracuje z wszystkimi najważniejszymi przeglądarkami internetowymi, ale Firefox

[4] *https://www.pythonanywhere.com/*

pozostaje najłatwiejszą w użyciu, ponieważ jest dostępna na różne platformy sprzętowe. Dodatkowym atutem pozostaje mniejsza podatność twórców przeglądarki Firefox na żądania korporacji.

System kontroli wersji Git
: Ten system jest dostępny dla dowolnej platformy sprzętowej i znajdziesz go w witrynie *http://git-scm.com/*.

Narzędzie pip przeznaczone do zarządzania pakietami Pythona
: Wymienione narzędzie zostało dołączone do Pythona 3.4 (nie zawsze znajdowało się w pakiecie, więc to zmiana na plus). Aby upewnić się że używamy narzędzia `pip` w wersji dla Pythona 3, w zaprezentowanych przykładach zawsze stosuję polecenie `pip3`. W zależności od platformy sprzętowej to może być również `pip-3.4` lub `pip-3.3`. Informacje szczegółowe na ten temat znajdziesz w ramkach poświęconych poszczególnym systemom operacyjnym.

Informacje dla użytkowników Windows

Użytkownicy Windows mogą czasami czuć się zaniedbywani, ponieważ systemy OS X i Linux pomagają zapomnieć o istnieniu świata poza paradygmatem systemów pochodnych dla systemu Unix. Lewe ukośniki jako separatory katalogów? Litery oznaczające napędy? Co takiego? Mimo wszystko materiał przedstawiony w książce może być wykorzystany także przez użytkowników systemu Windows. Oto garść podpowiedzi ułatwiających pracę.

1. Podczas instalacji systemu kontroli wersji Git w Windows upewnij się o wyborze opcji *Run Git and included Unix tools form the Windows command prompt*. W ten sposób uzyskasz dostęp do programu o nazwie Git Bash. Wykorzystaj go jako podstawowy wiersz poleceń, a będziesz mógł korzystać ze wszystkich użytecznych narzędzi powłoki GNU, takich jak `ls`, `touch` i `grep`. Bonusem będzie możliwość użycia zwykłych ukośników jako separatorów katalogów.
2. Podczas instalacji Pythona 3 upewnij się o zaznaczeniu opcji *Add python.exe to Path*, jak pokazano na rysunku 0.1. W ten sposób będziesz mógł uruchamiać Pythona z poziomu powłoki.

Rysunek 0.1. Dodanie Pythona do systemowej ścieżki dostępu

3. W systemie Windows plik wykonywalny Pythona 3 nosi nazwę *python.exe*. To jest dokładnie taka sama nazwa jak w przypadku Pythona 2. W celu uniknięcia ewentualnych niejasności w katalogu plików binarnych Git Bash utwórz następujące dowiązanie symboliczne:

   ```
   ln -s /c/Python34/python.exe /bin/python3.exe
   ```

   Może wystąpić konieczność kliknięcia Git Bash prawym przyciskiem myszy i wybrania opcji *Uruchom jako administrator*. Zwróć uwagę, że utworzone tutaj dowiązanie symboliczne działa jedynie w powłoce Git Bash, a nie w zwykłym wierszu poleceń systemu Windows.

4. Wydanie Python 3.4 jest dostarczane wraz z narzędziem `pip`. Możesz sprawdzić, czy narzędzie `pip` zostało zainstalowane, wydając polecenie `which pip3`. Wynikiem wykonania wymienionego polecenia powinno być `/c/Python34/Scripts/pip3`.

   Jeżeli z jakiegokolwiek powodu musisz korzystać z wydania Python 3.3 i nie masz narzędzia `pip3`, informacje dotyczące jego instalacji znajdziesz w witrynie *https://pip.pypa.io/en/latest/*. W trakcie pisania książki procedura instalacji wymagała pobrania pliku, a następnie uruchomienia go za pomocą polecenia `python3 get-pip.py`. *Upewnij się o użyciu polecenia* `python3` *podczas uruchamiania skryptu konfiguracyjnego*.

Po uwzględnieniu powyższych wskazówek powinieneś mieć możliwość przejścia do powłoki Git Bash i wydania poleceń `python3` i `pip3` z poziomu dowolnego katalogu.

Informacje dla użytkowników OS X

Wprawdzie system OS X jest lepiej niż Windows przystosowany do obsługi Pythona, ale do niedawna instalacja narzędzia `pip3` była niełatwym zadaniem. Wraz z wydaniem wersji Python 3.4 wszystko uległo zmianie i proces instalacji jest wręcz trywialny.

- Wydanie Python 3.4 można zainstalować bez żadnych problemów za pomocą *dostępnego instalatora*[5]. Narzędzie `pip` zostanie zainstalowane automatycznie.
- Instalator systemu kontroli wersji Git również działa bez problemów.

Podobnie jak w przypadku Windows po przeprowadzeniu instalacji oprogramowania będziesz mógł otworzyć okno powłoki i z poziomu dowolnego katalogu wydawać polecenia `git`, `python3` i `pip3`. Jeżeli wystąpią jakiekolwiek problemy, w ulubionej wyszukiwarce internetowej wpisz „ścieżka systemowa" lub „polecenie nieznalezione", a otrzymasz listę wielu zasobów, w których znajdziesz informacje niezbędne do rozwiązania problemu.

Warto również zwrócić uwagę na menedżera pakietów *Homebrew*[6], za pomocą którego w systemie OS X można w niezawodny sposób zainstalować wiele narzędzi pochodzących z systemu Unix (w tym również Pythona 3). Wprawdzie oficjalny instalator Pythona działa już bez problemów, ale wymieniony menedżer może okazać się użyteczny w przyszłości. Jego użycie wymaga pobrania Xcode (jego wielkość to obecnie około 2,5 GB), ale w ten sposób zyskujesz kompilator C, co będzie przydatnym „efektem ubocznym".

[5] *https://www.python.org/downloads/mac-osx/*

[6] *http://brew.sh/*

Domyślny edytor Git oraz pozostała podstawowa konfiguracja Git

W dalszej części książki przedstawię polecenia pokazujące krok po kroku użycie Git, choć dobrym rozwiązaniem będzie przeprowadzenie już teraz pewnej konfiguracji. Na przykład w trakcie pierwszego przekazywania plików do repozytorium zostanie wyświetlone okno edytora `vi`. W takiej sytuacji być może nie będziesz wiedział, co należy dalej zrobić. Cóż, edytor `vi` ma dwa tryby pracy, więc masz dwa wyjścia do wyboru. Pierwsze polega na poznaniu kilku najpotrzebniejszych poleceń edytora (*naciśnij klawisz **i** przechodząc w ten sposób do trybu wstawiania, wprowadź odpowiedni tekst; naciśnij **Esc**, aby powrócić do zwykłego trybu edytora, a następnie zapisz plik i zakończ pracę z* `vi`, *wpisując* `:wq` *i naciskając **Enter***). W ten sposób dołączysz do wspaniałego środowiska osób, które znają ten stary i niezwykle ceniony edytor.

Drugie rozwiązanie polega na zupełnym odrzuceniu jakiegokolwiek powrotu do lat siedemdziesiątych ubiegłego wieku i skonfigurowaniu Git do użycia innego, dowolnie wybranego edytora. Zamknij `vi`, naciskając *Esc* i następnie wpisując `:q!`, a następnie zmień domyślny edytor Git. W dokumentacji Git znajdziesz informacje dotyczące *podstawowej konfiguracji Git*[7].

Wymagane moduły Pythona

Po zainstalowaniu menedżera `pip` instalacja nowych modułów Pythona jest niezwykle łatwym zadaniem. Nowe moduły będziemy instalować, gdy okażą się niezbędne. Jednak już kilka powinniśmy mieć zainstalowanych na samym początku.

- *Django 1.7* — **`sudo pip3 install django==1.7`** (w systemie Windows pomiń `sudo`). Django to wybrany przez nas framework sieciowy. Upewnij się o instalacji wersji 1.7. Dzięki wymienionemu modułowi uzyskasz dostęp do django-admin.py z poziomu powłoki. Jeżeli potrzebujesz pomocy, pewne informacje dotyczące instalacji znajdziesz w *dokumentacji Django*[8].

W maju 2014 roku framework Django 1.7 nadal pozostawał w wersji beta. Jeżeli wymienione powyżej polecenie nie działa, użyj następującego: **`sudo pip3 install https://github.com/djan go/django/archive/stable/1.7.x.zip`**.

- *Selenium* — **`sudo pip3 install --upgrade selenium`** (w systemie Windows pomiń `sudo`). Selenium to narzędzie automatyzacji pozwalające nam na wykonywanie tak zwanych testów funkcjonalnych. Upewnij się o instalacji absolutnie najnowszej dostępnej wersji narzędzia. Selenium bierze udział w nieustającym wyścigu z najważniejszymi przeglądarkami internetowymi, próbując zachować zgodność z najnowszymi funkcjami, które są w nich implementowane. Jeżeli kiedykolwiek zauważysz dziwne zachowanie Selenium, powodem będzie instalacja nowej wersji przeglądarki Firefox. W takim przypadku rozwiązaniem będzie uaktualnienie do najnowszej wersji Selenium…

O ile doskonale nie wiesz, co robisz, *nie* używaj narzędzia `virtualenv`. Tego rodzaju narzędzie zaczniemy wykorzystywać w rozdziale 8.

[7] *http://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration*

[8] *https://docs.djangoproject.com/en/1.7/intro/install/*

Informacje dotyczące IDE

Jeżeli masz doświadczenie w programowaniu w języku Java lub .NET, to być może myślisz o wykorzystaniu środowiska IDE podczas pracy nad projektami w Pythonie. Wspomniane środowiska oferują wiele użytecznych narzędzi, w tym również integrację z VCS. Na rynku dostępnych jest sporo doskonałych rozwiązań IDE przeznaczonych dla Pythona. Kiedy zaczynałem pracę z Pythonem, sam zacząłem używać środowiska IDE i uznawałem je za niezwykle użyteczne w kilku pierwszych projektach.

Jednak sugeruję (to jedynie propozycja) *rezygnację* z użycia środowiska IDE, przynajmniej podczas lektury niniejszej książki. W świecie Pythona nie istnieje aż tak duża potrzeba użycia IDE. Tę książkę napisałem z założeniem, że będziesz korzystał z prostego edytora tekstów oraz powłoki. Czasami do dyspozycji masz właśnie tylko tyle — na przykład podczas pracy z serwerem. Dlatego też warto nauczyć się użycia podstawowych narzędzi i poznać sposoby ich działania. W ten sposób zdobędziesz przydatną wiedzę, nawet jeśli po zakończeniu lektury książki postanowisz powrócić do środowiska IDE i wszystkich oferowanych przez nie użytecznych narzędzi.

Jeżeli przedstawione tutaj informacje nie sprawdzają się lub chcesz zaproponować lepsze rozwiązania, zawsze możesz do mnie napisać na adres *obeythetestinggoat@gmail.com*.

Podziękowania

Pragnę podziękować wielu osobom, bez których ta książka nigdy by nie powstała lub byłaby gorsza niż w obecnej postaci.

Przede wszystkim dziękuję „Gregowi" z $OTHER_PUBLISHER, który był pierwszą osobą zachęcającą do uwierzenia, że tę książkę naprawdę można napisać. Wprawdzie Twoi pracownicy okazali się mieć przesadnie rygorystyczne podejście do praw autorskich, ale jestem Ci niezmiernie wdzięczny za wiarę we mnie.

Podziękowania składam Michaelowi Foordowi, kolejnemu byłemu pracownikowi Resolver Systems, za początkową inspirację do napisania książki oraz nieustanną pomoc podczas realizacji tego projektu. Dziękuję także mojemu szefowi Gilesowi Thomasowi, że naiwnie pozwolił kolejnemu pracownikowi na napisanie książki. (Jestem przekonany, że teraz już zmienił tekst standardowej umowy o pracę i zamieścił w niej klauzulę „żadnych książek"). Dziękuję Ci za nieustającą mądrość i skierowanie mnie na ścieżkę testów.

Dziękuję moim pozostałym kolegom, Glennowi Jonesowi i Hanselowi Dunlopowi, za nieocenioną pomoc i cierpliwość na przestrzeni ostatniego roku.

Dziękuję mojej żonie Clementine oraz obu rodzinom — bez ich wsparcia i cierpliwości nigdy nie napisałbym tej książki. Jednocześnie przepraszam Was za cały czas, który spędziłem z nosem w komputerze w chwilach, które powinny być jednak niezapomnianymi chwilami rodzinnymi. Nie miałem pojęcia, jak ta książka wpłynie na moje życie („Chcesz ją napisać w wolnych chwilach? Brzmi rozsądnie…"). Clementine, nie zrobiłbym tego bez Ciebie.

Dziękuję recenzentom technicznym Jonathanowi Hartleyowi, Nicholasowi Tollerveyowi i Emily Bache za słowa zachęty i nieocenione komentarze. Podziękowania kieruję zwłaszcza pod adresem Emily, która uważnie przeczytała wszystkie rozdziały. Oczywiście jestem także niezmiernie wdzięczny Nickowi i Jonowi. Dzięki Waszej obecności nie miałem poczucia, że pracuję zupełnie sam nad tym projektem. Bez pomocy wymienionych recenzentów technicznych ta książka byłaby tylko mniej lub bardziej sensownym wywodem idioty.

Dziękuję również każdemu, kto poświęcił swój czas na dostarczanie informacji zwrotnych o książce, nawet w postaci kilku dobrych życzeń; są to: Gary Bernhardt, Mark Lavin, Matt O'Donnell, Michael Foord, Hynek Schlawack, Russell Keith-Magee, Andrew Godwin i Kenneth Reitz. Dziękuję, że okazaliście się sprytniejsi ode mnie i uchroniliście mnie przed napisaniem wielu głupstw. Oczywiście w książce pozostało wiele innych głupstw, za które ponoszę wyłączną odpowiedzialność.

Dziękuję mojej redaktor Meghan Blanchette za okazaną przyjaźń i sympatię, za kierowanie projektem w zakresie zarówno ram czasowych, jak i ograniczania moich kolejnych głupich pomysłów. Dziękuję pozostałym pracownikom wydawnictwa O'Reilly za pomoc, między innymi Sarze Schneider, Karze Ebrahim i Danowi Fauxsmithowi za możliwość pozostania przy języku angielskim, którym posługujemy się w Wielkiej Brytanii. Dziękuję Charlesowi Roumeliotisowi za pomoc nad stylem i gramatyką. Wprawdzie możesz nigdy nie dostrzec meritum chicagowskiej szkoły stosowania znaków cytowania i punktowania, ale jestem pewien, że cieszysz się z jej istnienia. Dziękuję także działowi graficznemu za umieszczenie kozy na okładce książki!

Specjalne podziękowania kieruję do wszystkich czytelników wczesnej wersji książki za pomoc w wychwytywaniu literówek, informacje, sugestie oraz wszelkie inne sposoby, na jakie pomogliście w usprawnieniu książki. Przede wszystkim jestem wdzięczny za ciepłe słowa zachęty i nieustające wsparcie. Osoby, którym dziękuję szczególnie, to: Jason Wirth, Dave Pawson, Jeff Orr, Kevin De Baere, crainbf, dsisson, Galeran, Michael Allan, James O'Donnell, Marek Turnovec, SoonerBourne, julz, Cody Farmer, William Vincent, Trey Hunner, David Souther, Tom Perkin, Sorcha Bowler, Jon Poler, Charles Quast, Siddhartha Naithani, Steve Young, Roger Camargo, Wesley Hansen, Johansen Christian Vermeer, Ian Laurain, Sean Robertson, Hari Jayaram, Bayard Randel, Konrad Korżel, Matthew Waller, Julian Harley, Barry McClendon, Simon Jakobi, Angelo Cordon, Jyrki Kajala, Manish Jain, Mahadevan Sreenivasan, Konrad Korżel, Deric Crago, Cosmo Smith, Markus Kemmerling, Andrea Costantini, Daniel Patrick, Ryan Allen, Jason Selby, Greg Vaughan, Jonathan Sundqvist, Richard Bailey, Diane Soini, Dale Stewart, Mark Keaton, Johan Wärlander, Simon Scarfe, Eric Grannan, Marc-Anthony Taylor, Maria McKinley, John McKenna i wiele innych. Jeżeli pominąłem Cię tutaj, masz absolutne prawo do złości. Jestem niezwykle wdzięczny także Tobie, napisz do mnie, a postaram się jakoś naprawić moje przeoczenie.

Na końcu dziękuję czytelnikom za decyzję o sprawdzeniu tej książki. Mam nadzieję, że jej lektura okaże się przyjemnością!

CZĘŚĆ I

Podstawy TDD i Django

W tej części poznasz podstawy *programowania sterowanego testami* (ang. *test-driven development*, TDD). Całkowicie od podstaw opracujemy prawdziwą aplikację sieciową i na każdym etapie prac będziemy przygotowywać dla niej testy.

Dokładnie omówiony będzie temat testowania funkcjonalnego za pomocą Selenium i testy jednostkowe, a także poznasz występujące między nimi różnice. Przedstawię tutaj sposób pracy w trakcie programowania sterowanego testami, który określam mianem cyklu „test jednostkowy i tworzenie kodu". Przeprowadzimy również refaktoring i zobaczymy, jak go stosować z technikami TDD. Ponieważ do tworzenia oprogramowania podchodzę niezwykle poważnie, podczas pracy będziemy używać systemu kontroli wersji (Git). Dowiesz się więc, jak i kiedy przekazywać kod do repozytorium oraz integrować go z technikami TDD i sposobem pracy w trakcie programowania sieciowego.

W książce wykorzystamy Django, czyli (prawdopodobnie) najpopularniejszy na świecie framework sieciowy dla Pythona. Postaram się powoli i pojedynczo wprowadzać nowe koncepcje Django oraz podać wiele łączy prowadzących do innych źródeł. Jeżeli nie miałeś wcześniej styczności z Django, to gorąco zachęcam Cię do zapoznania się z materiałem przedstawionym we wspomnianych źródłach. Jeżeli poczujesz się nieco zagubiony w trakcie lektury książki, zrób sobie przerwę, poświęć kilka godzin na przejrzenie oficjalnych samouczków Django, a następnie powróć do lektury książki.

Oczywiście spotkasz również Testing Goat…

Zachowaj ostrożność podczas kopiowania i wklejania

Jeżeli czytasz książkę w wersji elektronicznej, naturalne jest, że będziesz chciał kopiować listingi i wklejać je w używanym środowisku pracy. Jednak znacznie lepszym rozwiązaniem będzie, jeśli zrezygnujesz z kopiowania kodu. Samodzielne wprowadzanie kodu zmusza mózg do efektywniejszego działania, a ponadto takie rozwiązanie jest bliższe rzeczywistemu sposobowi tworzenia kodu. Po drodze niewątpliwie popełnisz pewne błędy, a ich samodzielne wyszukanie okaże się cenną lekcją.

Poza wszystkim warto również pamiętać o niedoskonałościach formatu PDF. Oznacza to, że podczas kopiowania i wklejania kodu mogą dziać się różne dziwne rzeczy…

ROZDZIAŁ 1.

Konfiguracja Django za pomocą testu funkcjonalnego

Programowanie sterowane testami nie jest czymś, co przychodzi naturalnie. To rodzaj dyscypliny, jak sztuki walki. Podobnie jak w filmie kung-fu będziesz potrzebował wymagającego mistrza, który zmusi Cię do zachowania dyscypliny. W naszym przypadku to będzie Testing Goat.

Słuchaj Testing Goat! Nie rób nic, dopóki nie przygotujesz testu

Wspomniany Testing Goat[1] to nieoficjalna maskotka technik TDD w społeczności Pythona i prawdopodobnie ma różne znaczenie dla poszczególnych osób. Dla mnie Testing Goat to głos wewnątrz, który trzyma mnie w pionie i nakazuje pozostać na „właściwej ścieżce testowania". Przypomina małe aniołki lub demony, które w kreskówkach pojawiają się na barkach bohaterów, ale mają niewiele zmartwień. Mam nadzieję, że dzięki niniejszej książce Testing Goat pojawi się także i w Twojej głowie.

Postanowiliśmy zbudować witrynę internetową, nawet jeśli jeszcze nie całkiem dokładnie wiemy, co należy zrobić. W świecie programowania sieciowego pierwszym krokiem jest wybór frameworka, jego instalacja i konfiguracja. *Pobierz to, zainstaluj tak, skonfiguruj to, uruchom skrypt…* Jednak programowanie sterowane testami wymaga zupełnie innego podejścia. Kiedy stosujesz techniki TDD, zawsze powinieneś mieć w sobie Testing Goat wołający „najpierw test, najpierw test!" i uparty niczym koza.

W programowaniu sterowanym testami krok pierwszy zawsze jest taki sam: *utwórz test*.

Dlatego też *najpierw* tworzymy test, *następnie* go wykonujemy i sprawdzamy, czy zgodnie z oczekiwaniami zakończył się niepowodzeniem. *Tylko wtedy* możemy przejść dalej, czyli do tworzenia aplikacji. Powinieneś o tym pamiętać i nieustannie sobie powtarzać.

Kolejną cechą kozy jest wykonywanie pojedynczych kroków. Dlatego też kozy rzadko spadają z gór, niezależnie od tego, jak strome mogą one być (patrz rysunek 1.1).

[1] Wymieniony zwrot dosłownie oznacza testującą kozę — *przyp. tłum*.

Rysunek 1.1. Kozy są o wiele zwinniejsze, niż sądzisz (źródło: Caitlin Stewart, Flickr)[2]

Będziemy poruszać się eleganckimi, małymi krokami. Do zbudowania aplikacji wykorzystamy Django, czyli popularny framework sieciowy dla Pythona.

Pierwszym zadaniem jest sprawdzenie, czy w środowisku pracy mamy zainstalowany framework Django oraz czy jest on gotowy do pracy. Wspomniane *sprawdzenie* będzie polegało na próbie uruchomienia serwera Django, aby przekonać się, czy udostępnia on stronę internetową w przeglądarce uruchomionej w komputerze lokalnym. Do wymienionego zadania wykorzystamy narzędzie automatyzacji o nazwie *Selenium*.

W katalogu przeznaczonym na budowany projekt utwórz nowy plik Pythona o nazwie *functional_tests.py*, a następnie umieść w nim przedstawiony poniżej kod. Jeżeli masz wrażenie, że słyszysz głosy kilku małych kóz, to może być pomocne.

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('http://localhost:8000')

assert 'Django' in browser.title
```

[2] *https://www.flickr.com/photos/caitlinstewart/2846642630/*

Pożegnanie z liczbami rzymskimi!

W wielu wprowadzeniach do technik TDD stosowane są przykłady liczb rzymskich, co wydaje się być żartem. Nawet ja się temu poddałem. Jeżeli jesteś ciekaw, możesz się o tym przekonać, odwiedzając *moją stronę w serwisie Github*[3].

Użycie przykładu opartego na liczbach rzymskich ma zarówno dobre, jak i złe strony. To jest przykład teoretycznego problemu o ograniczonym zakresie, a na jego podstawie można dość dobrze wyjaśnić techniki TDD.

Problem polega na tym, że trudno to później powiązać z rzeczywistymi rozwiązaniami. Dlatego też zdecydowałem się na opracowanie zupełnie od zera przykładowej aplikacji sieciowej. Wprawdzie to będzie bardzo prosta aplikacja, ale mam nadzieję, że ułatwi Ci ona później przejście do kolejnego, rzeczywistego projektu.

Przedstawiony powyżej kod to nasz pierwszy *test funkcjonalny* (ang. *functional test*). W dalszej części książki dowiesz się więcej na temat testów funkcjonalnych oraz cech odróżniających je od testów jednostkowych. Teraz wystarczy upewnić się o zrozumieniu sposobu działania przedstawionego testu:

- Uruchomienie modułu *webdriver* narzędzia Selenium w celu wyświetlenia okna przeglądarki internetowej Firefox.
- Użycie wymienionego modułu do wyświetlenia strony internetowej, która powinna być pobrana z komputera lokalnego.
- Sprawdzenie (za pomocą asercji), czy tytuł wczytanej strony internetowej zawiera słowo *Django*.

Spróbujmy uruchomić przygotowany plik:

```
$ python3 functional_tests.py
Traceback (most recent call last):
  File "functional_tests.py", line 6, in <module>
    assert 'Django' in browser.title
AssertionError
```

Na ekranie powinieneś zobaczyć okno przeglądarki internetowej, w którym następuje próba przejścia do adresu *localhost:8000*. Następnie wyświetlany jest komunikat błędu Pythona. Prawdopodobnie będziesz poirytowany pozostawionym na ekranie oknem przeglądarki Firefox. Tym zajmiemy się nieco później.

Jeżeli zamiast wspomnianego powyżej komunikatu otrzymasz informację o błędzie dotyczącym importu narzędzia Selenium, powinieneś powrócić do „Wprowadzenia" i raz jeszcze zapoznać się z podrozdziałem omawiającym wymagania dotyczące oprogramowania, jakie powinieneś mieć zainstalowane w komputerze.

Na tym etapie mamy *test kończący się niepowodzeniem*, co oznacza, że możemy przystąpić do budowy naszej aplikacji.

[3] *https://github.com/hjwp/*

Rozpoczęcie pracy z frameworkiem Django

Ponieważ jesteś już po lekturze przedstawionego we „Wprowadzeniu" podrozdziału poświęconego wymaganiom i przyjętym założeniom, to powinieneś mieć zainstalowany framework Django. Pierwszym krokiem podczas rozpoczęcia pracy z Django jest utworzenie *projektu*, czyli kontenera dla naszej witryny internetowej. Do tego celu Django udostępnia niewielkie narzędzie działające w powłoce:

```
$ django-admin.py startproject superlists
```

Wynikiem wydania powyższego polecenia będzie utworzenie katalogu o nazwie *superlists*, zawierającego pewną liczbę podkatalogów i plików:

```
.
    functional_tests.py
    superlists
        manage.py
        superlists
            __init__.py
            settings.py
            urls.py
            wsgi.py
```

Jak możesz zobaczyć, katalog projektu *superlists* zawiera również podkatalog o tej samej nazwie. Wprawdzie to może być nieco mylące, ale to drobiazg. Jeżeli chcesz wiedzieć, skąd wziął się podkatalog o nazwie takiej samej jak projekt, będziesz się musiał cofnąć do historii Django. W tym momencie najważniejsze jest, aby zapamiętać, że katalog *superlists/superlists* jest przeznaczony dla komponentów stosowanych w całym projekcie. Przykładem może być plik *settings.py*, który jest używany do przechowywania konfiguracji globalnej witryny internetowej.

Zwróć również uwagę na plik o nazwie *manage.py*. To wszechstronne narzędzie Django, wykorzystasz je do uruchomienia serwera. Wypróbujmy je teraz. Wydaj polecenie **`cd superlists`**, aby przejść do katalogu projektu *superlists* (na poziomie tego katalogu będziemy wykonywali wiele zadań). Następnie wydaj poniższe polecenie:

```
$ python3 manage.py runserver
Validating models...

0 errors found
Django version 1.7, using settings 'superlists.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

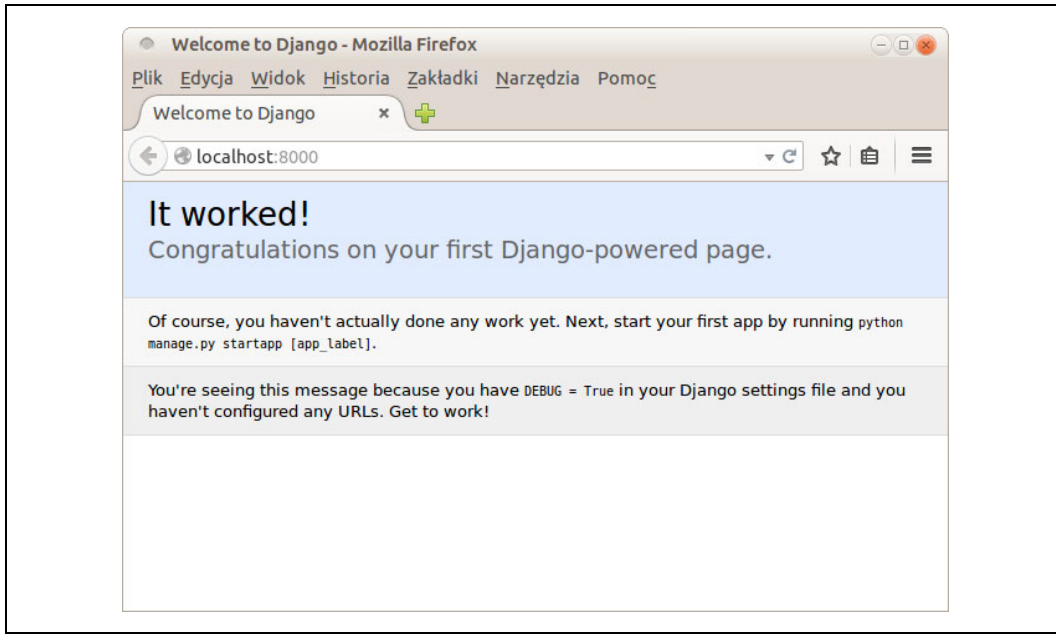
Pozostaw uruchomiony serwer i otwórz kolejne okno powłoki. W nowo otwartym oknie ponownie spróbuj wykonać nasz test (z poziomu katalogu, w którym znajduje się plik *functional_tests.py*):

```
$ python3 functional_tests.py
$
```

W powłoce nie zobaczysz nic ciekawego, ale powinieneś zwrócić uwagę na dwie kwestie. Pierwsza to brak niemiłego komunikatu `AssertionError`. Natomiast druga to wyświetlone przez narzędzie Selenium okno przeglądarki internetowej Firefox wyświetlające dziwnie wyglądającą stronę.

Strona nie prezentuje się zbyt zachęcająco, ale jest to nasz pierwszy zaliczony test. Hura!

Jeżeli masz wrażenie, że to jest magia, spójrz na uruchomiony serwer. W tym celu samodzielnie otwórz nowe okno przeglądarki internetowej i przejdź pod adres *http://localhost:8000/*. Powinieneś otrzymać wynik podobny do pokazanego na rysunku 1.2.



Rysunek 1.2. Serwer frameworka Django działa

Możesz teraz zakończyć działanie serwera. Wystarczy powrócić do początkowego okna powłoki i nacisnąć klawisze *Ctrl+C*.

Utworzenie repozytorium Git

Zanim zakończymy ten rozdział, do wykonania pozostało nam jeszcze jedno zadanie — umieszczenie w systemie kontroli wersji (ang. *version control system*, VCS) utworzonego dotąd kodu. Jeżeli jesteś doświadczonym programistą, nie muszę Cię przekonywać o zaletach systemu kontroli wersji. Natomiast jeśli dopiero rozpoczynasz przygodę z programowaniem, to musisz mi uwierzyć na słowo, że system kontroli wersji to po prostu konieczność. Gdy nad projektem pracujesz przez kilka tygodni i zawiera on setki wierszy kodu, wówczas narzędzie pozwalające na przeglądanie jego wcześniejszych wersji, wycofywanie wprowadzonych zmian czy bezpieczne wypróbowanie nowych koncepcji jest wręcz nieocenione, a kod umieszczony w VCS stanowi rodzaj kopii zapasowej. Techniki TDD doskonale sprawdzają się w połączeniu z systemem kontroli wersji i dlatego chcę pokazać, jak system VCS można wykorzystać we własnym środowisku pracy.

Najwyższa pora na umieszczenie pierwszego kodu w repozytorium. Jeżeli to nieco za późno, możesz mnie za to winić. W książce będziemy używać systemu kontroli Git, ponieważ uznaję go za najlepszy z dostępnych.

Rozpoczynamy od przeniesienia pliku *functional_tests.py* do podkatalogu *superlists*, a następnie wydamy polecenie `git init` w celu zainicjowania repozytorium.

```
$ ls
superlists          functional_tests.py
$ mv functional_tests.py superlists/
$ cd superlists
$ git init .
Initialised empty Git repository in /workspace/superlists/.git/
```

Od tej chwili katalog główny *superlists* staje się naszym katalogiem roboczym. Gdy zobaczysz polecenie do wykonania, możesz przyjąć założenie, że powinno być wydane w wymienionym katalogu. Podobnie jeśli w tekście podaję ścieżkę dostępu do pliku, potraktuj ją jako względną dla katalogu głównego *superlists*. Dlatego też *superlists/settings.py* oznacza plik *settings.py* znajdujący się w podkatalogu *superlists* katalogu głównego o tej samej nazwie. Czy wszystko jasne? Jeżeli będziesz miał jakiekolwiek wątpliwości, szukaj pliku *manage.py*, ponieważ powinieneś być w tym samym katalogu, w którym jest wymieniony plik.

Teraz wybierzemy pliki, które mają zostać umieszczone w repozytorium — w tym przypadku to prawie wszystkie pliki.

```
$ ls
db.sqlite3  manage.py   superlists  functional_tests.py
```

Plik o nazwie *db.sqlite3* jest plikiem bazy danych. Ponieważ nie chcemy go umieszczać w systemie kontroli wersji, do projektu dodajemy plik *.gitignore* wskazujący pliki, które mają być ignorowane przez Git:

```
$ echo "db.sqlite3" >> .gitignore
```

W tym momencie pozostałą zawartość katalogu bieżącego (wskazywanego przez kropkę) możemy dodać do repozytorium:

```
$ git add .
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   functional_tests.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/__pycache__/__init__.cpython-34.pyc
        new file:   superlists/__pycache__/settings.cpython-34.pyc
        new file:   superlists/__pycache__/urls.cpython-34.pyc
        new file:   superlists/__pycache__/wsgi.cpython-34.pyc
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py
```

Psiakrew! W projekcie znajduje się mnóstwo plików *.pyc*, których umieszczanie w repozytorium jest bezcelowe. Musimy je więc usunąć z repozytorium oraz dodać do listy w pliku *.gitignore*:

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-34.pyc'
rm 'superlists/__pycache__/settings.cpython-34.pyc'
rm 'superlists/__pycache__/urls.cpython-34.pyc'
rm 'superlists/__pycache__/wsgi.cpython-34.pyc'
$ echo "__pycache__" >> .gitignore
$ echo "*.pyc" >> .gitignore
```

Przekonajmy się, co dotąd osiągnęliśmy… (Jak możesz zobaczyć, dość często korzystam z polecenia `git status`. Ponieważ wydaję je naprawdę często, zdefiniowałem dla niego alias `git st`. Oczywiście nie zmuszam Cię do tego samego i dlatego zachęcam do samodzielnego definiowania własnych aliasów systemu kontroli Git).

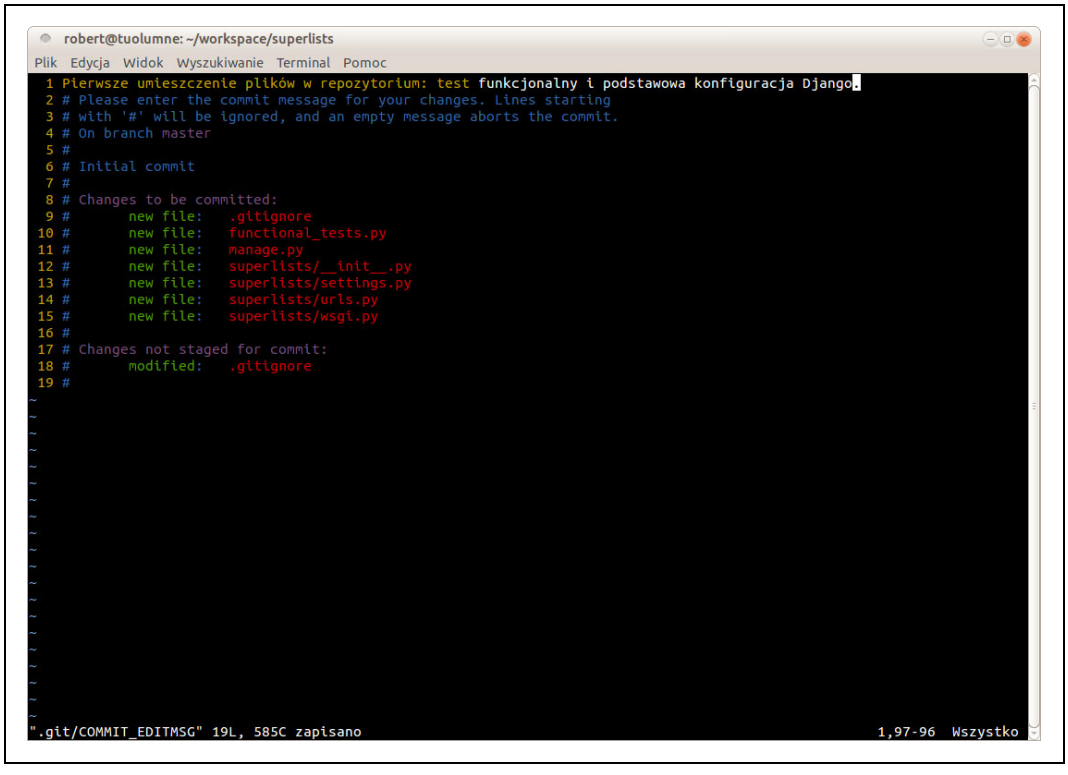
```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   functional_tests.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   .gitignore
```

Wszystko wygląda dobrze, więc jesteśmy gotowi do pierwszego umieszczenia plików w repozytorium:

```
$ git commit
```

Po wydaniu powyższego polecenia zostanie wyświetlone okno edytora[4] (podobne do pokazanego na rysunku 1.3) pozwalające na wprowadzenie opisu dotyczącego plików umieszczanych w repozytorium.

[4] Czy na ekranie zobaczyłeś okno edytora vi i zupełnie nie wiesz, co należy teraz zrobić? A może wyświetlony został komunikat dotyczący konta użytkownika i polecenie `git config --global user.username`? W takiej sytuacji powróć do przedstawionego we „Wprowadzeniu" podrozdziału poświęconego wymaganiom dotyczącym niezbędnego oprogramowania zainstalowanego w komputerze lokalnym. Znajdziesz tam kilka podpowiedzi pomocnych w wymienionych powyżej sytuacjach.



Rysunek 1.3. Pierwsze umieszczenie plików w repozytorium

Jeżeli naprawdę chcesz poznać Git, warto w tym miejscu poświęcić nieco czasu na opanowanie umiejętności przekazywania plików do repozytoriów umieszczonych w chmurze, oferowanych przez serwisy takie jak GitHub i BitBucket. Wykorzystanie repozytorium umieszczonego w chmurze będzie użyteczne, jeśli podczas lektury książki będziesz pracował na więcej niż tylko jednym komputerze. Opanowanie umiejętności pracy ze zdalnymi repozytoriami pozostawiam już Tobie, wymienione serwisy oferują w tym zakresie doskonałą dokumentację. Ewentualnie możesz się z tym wstrzymać aż do rozdziału 8., w którym zdalne repozytorium wykorzystamy do wdrożenia naszego projektu.

To już wszystko w kwestii systemu kontroli wersji. Gratulacje! Utworzyłeś test funkcjonalny za pomocą narzędzia Selenium, a także rozpocząłeś pracę z frameworkiem Django w sposób certyfikowany i zgodny z filozofią TDD. Zrób sobie zasłużony odpoczynek, zanim przejdziesz do lektury rozdziału 2.

ROZDZIAŁ 2.

Rozszerzenie testu funkcjonalnego za pomocą modułu unittest

Przystąpimy teraz do modyfikacji naszego testu, który na obecnym etapie po prostu sprawdza istnienie domyślnej strony Django. Zamiast tego chcemy wyszukać tekst, który będzie znajdował się na stronie głównej naszej witryny.

Najwyższy czas na ujawnienie rodzaju budowanej tutaj aplikacji sieciowej — to lista rzeczy do zrobienia. W ten sposób wpisujemy się w obecną modę. Kilka lat temu ogromna część samouczków z zakresu programowania sieciowego była poświęcona budowie bloga. Następnie mieliśmy fora i ankiety. Teraz nadeszła pora na listy rzeczy do zrobienia.

Warto w tym miejscu dodać, że lista rzeczy do zrobienia to naprawdę użyteczny przykład. W zasadzie to bardzo prosta konstrukcja — lista ciągów tekstowych — co niezwykle ułatwia przygotowanie minimalnej, działającej wersji aplikacji. Jednak tego rodzaju aplikację można rozbudować na wiele różnych sposobów, na przykład zastosować odmienne modele trwałych magazynów danych, dodać terminy wykonania poszczególnych zadań, przypomnienia, zapewnić możliwość współdzielenia danych z innymi użytkownikami, a także usprawnić interfejs użytkownika wyświetlany po stronie klienta. Nie ma żadnego powodu, aby ograniczać się jedynie do „listy rzeczy do zrobienia", to może być dowolnego rodzaju lista. Moim celem jest pokazanie wszystkich najważniejszych aspektów programowania sieciowego oraz zastosowanie w nich technik TDD.

Użycie testu funkcjonalnego do przygotowania minimalnej aplikacji

Testy oparte na narzędziu Selenium pozwalają na użycie prawdziwej przeglądarki internetowej, a tym samym na sprawdzenie *funkcjonowania* aplikacji z perspektywy użytkownika. Dlatego też są nazywane *testami funkcjonalnymi*.

Oznacza to, że test funkcjonalny może być dowolnego rodzaju specyfikacją aplikacji. Jest przeznaczony do monitorowania tego, co można nazwać *informacjami od użytkownika*, i opiera się na sposobie, w jaki użytkownik może korzystać z danej funkcji, a także na tym, jak aplikacja powinna reagować na działania użytkownika.

Terminologia: test funkcjonalny == test akceptacji == test E2E

To, co ja nazywam testem funkcjonalnym, inni wolą określać mianem *testu akceptacji* lub testu *E2E*. Najważniejsze pozostaje to, że tego rodzaju testy sprawdzają z zewnątrz sposób działania całej aplikacji. Innym spotykanym tutaj pojęciem jest *test czarnej skrzynki*, ponieważ test nie ma żadnych informacji dotyczących wewnętrznych komponentów sprawdzanego systemu.

Testy funkcjonalne powinny zapewniać czytelne dla człowieka informacje, którymi można się kierować. Dodamy więc jasne komentarze do przygotowanego wcześniej kodu testu. Podczas tworzenia nowego testu funkcjonalnego możesz najpierw przygotować odpowiedni komentarz przeznaczony do przechwycenia kluczowych punktów informacji od użytkownika. Ponieważ testy funkcjonalne mają opisy czytelne dla człowieka, to można je przekazywać także osobom niebędącym programistami i w taki sposób dyskutować nad wymaganiami oraz funkcjami aplikacji.

Techniki TDD oraz metodologie programowania zwinnego bardzo często są stosowane razem. Jedną z częściej pojawiających się kwestii będzie to, jaka jest najprostsza możliwa wersja zapewniająca użyteczność aplikacji. Rozpoczynamy więc od jej utworzenia, aby jak najszybciej móc przystąpić do testowania.

Minimalna działająca wersja aplikacji listy rzeczy do zrobienia musi tak naprawdę pozwalać użytkownikowi na wprowadzanie rzeczy do zrobienia oraz przechowywać tę listę aż do kolejnego uruchomienia aplikacji przez użytkownika.

Otwórz plik *functional_tests.py* i zmodyfikuj go do przedstawionej poniżej postaci.

Plik *functional_tests.py*:

```
from selenium import webdriver

browser = webdriver.Firefox()

# Edyta dowiedziała się o nowej, wspaniałej aplikacji w postaci listy rzeczy do zrobienia.
# Postanowiła więc przejść na stronę główną tej aplikacji.
browser.get('http://localhost:8000')

# Zwróciła uwagę, że tytuł strony i nagłówek zawierają słowo Listy.
assert 'Listy' in browser.title

# Od razu zostaje zachęcona, aby wpisać rzecz do zrobienia.

# W polu tekstowym wpisała "Kupić pawie pióra" (hobby Edyty
# polega na tworzeniu ozdobnych przynęt).

# Po naciśnięciu klawisza Enter strona została uaktualniona i wyświetla
# "1: Kupić pawie pióra" jako element listy rzeczy do zrobienia.

# Na stronie nadal znajduje się pole tekstowe zachęcające do podania kolejnego zadania.
# Edyta wpisała "Użyć pawich piór do zrobienia przynęty" (Edyta jest niezwykle skrupulatna).

# Strona została ponownie uaktualniona i teraz wyświetla dwa elementy na liście rzeczy do zrobienia.

# Edyta była ciekawa, czy witryna zapamięta jej listę. Zwróciła uwagę na wygenerowany dla niej
# unikatowy adres URL, obok którego znajduje się pewien tekst z wyjaśnieniem

# Przechodzi pod podany adres URL i widzi wyświetloną swoją listę rzeczy do zrobienia.

# Usatysfakcjonowana kładzie się spać.

browser.quit()
```

Mamy określenie na komentarze...

Kiedy zacząłem pracę w firmie Resolver, tworzony przez siebie kod wręcz naszpikowałem opisowymi komentarzami. Moi współpracownicy powiedzieli mi: „Harry, mamy określenie na komentarze. Nazywamy je kłamstwami". Byłem zszokowany! Przecież w szkole dowiedziałem się, że umieszczanie komentarzy należy do dobrej praktyki.

Jednak nie należy z nimi przesadzać. W kodzie na pewno znajdzie się miejsce na komentarze, które mają wskazać kontekst i zamierzenia autora. Natomiast zupełnie bezcelowe jest umieszczanie komentarzy powtarzających to, co można bez problemu odczytać na podstawie kodu:

```
# Inkrementacja zmiennej wibble o 1.
wibble += 1
```

Powyższy komentarz nie tylko jest bezcelowy, ale również niebezpieczny. Jeżeli zapomnisz o jego uaktualnieniu po modyfikacji kodu, wówczas komentarz stanie się mylący. Idealnym rozwiązaniem jest dążenie do zapewnienia maksymalnej czytelności kodu, na przykład przez użycie dobrych nazw dla zmiennych i funkcji. W połączeniu z doskonale przemyślaną strukturą nie będziesz potrzebował żadnych komentarzy wyjaśniających *sposób* działania kodu. Wówczas w różnych miejscach wystarczy jedynie dodanie komentarzy wyjaśniających *powody* użycia danego kodu.

Istnieje jeszcze wiele innych miejsc, w których komentarze są bardzo użyteczne. Przekonasz się, że framework Django wstawia je w wielu plikach generowanych dla użytkownika i używa w charakterze podpowiedzi dotyczących API frameworka. Ponadto komentarze wykorzystujemy w celu wyjaśnienia otrzymanych informacji od użytkownika umieszczanych w testach funkcjonalnych. Wymuszając przygotowanie spójnych informacji dotyczących testu, zawsze mamy pewność, że test będzie przeprowadzony z uwzględnieniem punktu widzenia użytkownika.

To jest tylko wierzchołek góry lodowej. Mamy jeszcze BDD (ang. *behavior-driven development*) i testowanie za pomocą DSL, ale to są tematy na zupełnie inne książki.

Zwróć uwagę, że poza zapisem testów w postaci komentarzy zmieniłem także polecenie `assert`, które teraz wyszukuje słowa *Listy* zamiast *Django*. To oznacza, że teraz oczekujemy zakończenia testu niepowodzeniem. Spróbujmy więc wykonać nasz test.

Najpierw uruchom serwer:

```
$ python3 manage.py runserver
```

Następnie w innej powłoce wykonaj test:

```
$ python3 functional_tests.py
Traceback (most recent call last):
  File "functional_tests.py", line 10, in <module>
    assert 'Listy' in browser.title
AssertionError
```

W ten sposób otrzymaliśmy tak zwane *oczekiwane niepowodzenie*, które wbrew nazwie jest tak naprawdę dobrą wiadomością. Wprawdzie nie tak dobrą, jak zakończenie testu powodzeniem, ale przynajmniej doskonale znamy przyczynę niepowodzenia. Mamy więc potwierdzenie prawidłowego utworzenia testu.

Moduł unittest ze standardowej biblioteki Pythona

Istnieje kilka drobnych, choć irytujących problemów, z którymi prawdopodobnie będziemy musieli sobie poradzić. Pierwszy z nich to niezbyt użyteczny komunikat o treści `AssertionError`. Byłoby dobrze, gdyby test podał rzeczywisty tytuł okna przeglądarki internetowej. Ponadto pozostawienie okna przeglądarki Firefox po zakończeniu testu jest niepotrzebne, więc jest pożądane, aby wspomniane okno zostało automatycznie zamknięte.

Jedną z możliwości jest użycie drugiego parametru w słowie kluczowym `assert`, na przykład w następujący sposób:

```
assert 'Listy' in browser.title, "Tytuł okna przeglądarki: " + browser.title
```

Do zamknięcia niepotrzebnego nam okna przeglądarki internetowej Firefox możemy wykorzystać konstrukcję `try-finally`. Jednak wspomniane wcześniej problemy pojawiają się tak często w trakcie przeprowadzania testów, że opracowano dla nich gotowe rozwiązania. Umieszczono je w module `unittest` znajdującym się w standardowej bibliotece Pythona. Wykorzystajmy więc gotowe rozwiązania! Zmodyfikuj plik *functional_tests.py*, aby przedstawiał się, jak pokazano poniżej.

Plik *functional_tests.py*:

```
from selenium import webdriver
import unittest

class NewVisitorTest(unittest.TestCase):  #❶

    def setUp(self):  #❷
        self.browser = webdriver.Firefox()

    def tearDown(self):  #❸
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self):  #❹
        # Edyta dowiedziała się o nowej, wspanialej aplikacji w postaci listy rzeczy do zrobienia.
        # Postanowiła więc przejść na stronę główną tej aplikacji.
        self.browser.get('http://localhost:8000')

        # Zwróciła uwagę, że tytuł strony i nagłówek zawierają słowo Listy.
        self.assertIn('Listy', self.browser.title)  #❺
        self.fail('Zakończenie testu!')  #❻

        # Od razu zostaje zachęcona, aby wpisać rzecz do zrobienia.
        [...pozostałe komentarze jak we wcześniejszym przykładzie...]

if __name__ == '__main__':  #❼
    unittest.main(warnings='ignore')  #❽
```

Prawdopodobnie zwróciłeś uwagę na kilka kwestii.

❶ Testy zostały zorganizowane w klasy dziedziczące po `unittest.TestCase`.

❹ Część główna testu znajduje się w metodzie o nazwie `test_can_start_a_list_and_retrieve_it_later()`. Każda metoda o nazwie rozpoczynającej się od `test_` jest metodą testową i będzie wykonana przez silnik testów. W klasie możesz mieć więcej niż tylko jedną metodę tego typu. Dobrym rozwiązaniem będzie stosowanie opisowych nazw dla metod testowych.

❷❸ `setUp()` i `tearDown()` to metody specjalne wykonywane odpowiednio przed i po każdym teście. Wykorzystuję je do uruchomienia i zamknięcia przeglądarki internetowej. Zwróć uwagę, że przypominają one nieco konstrukcję `try-catch`, ponieważ metoda `tearDown()` zostanie wykonana nawet wtedy, gdy w trakcie testu wystąpi błąd[1]. Po zakończeniu testu na ekranie nie będą dłużej pozostawały niepotrzebne nam okna przeglądarki internetowej Firefox.

❺ Do przeprowadzenia asercji wykorzystujemy `self.assertIn` zamiast po prostu słowa kluczowego `assert`. Moduł `unittest` oferuje wiele funkcji pomocniczych przydatnych podczas stosowania asercji, na przykład `assertEqual`, `assertTrue`, `assertFalse` itd. Więcej informacji na ten temat znajdziesz w *dokumentacji modułu* `unittest`[2].

❻ Polecenie `self.fail` niezależnie od wszystkiego kończy się niepowodzeniem i wyświetla wskazany komunikat błędu. W omawianym kodzie to przypomnienie o zakończeniu testu.

❼ Na końcu mamy klauzulę `if __name__ == '__main__'` (jeśli wcześniej nie spotkałeś się z tego rodzaju kodem, to powinieneś wiedzieć, że za jego pomocą skrypt Pythona sprawdza, czy został uruchomiony z poziomu powłoki, a nie zaimportowany przez inny skrypt). Wywołujemy `unittest.main()`, co powoduje uruchomienie silnika testów `unittest`, który z kolei automatycznie wyszukuje w pliku klasy i metody testowe, a następnie je wykonuje.

❽ Argument `warnings='ignore'` powoduje zawieszenie wyświetlania zbędnego komunikatu `ResourceWarning` dodanego w trakcie powstawania tej książki. Być może zniknie on w chwili, gdy będziesz miał tę książkę w rękach — w takim przypadku możesz spokojnie usunąć z kodu wymieniony argument.

Jeżeli zapoznałeś się z poświęconą testom dokumentacją Django, mogłeś się natknąć na klasę `LiveServerTestCase` i być może zastanawiasz się, czy powinieneś jej używać. Należą Ci się brawa za lekturę podręcznika. Objaśnienie działania wymienionej klasy jest teraz zbyt skomplikowane, ale obiecuję, że wykorzystamy ją w późniejszych rozdziałach książki.

Wypróbujmy teraz zmodyfikowany test!

```
$ python3 functional_tests.py
F
======================================================================
FAIL: test_can_start_a_list_and_retrieve_it_later (__main__.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "functional_tests.py", line 18, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('Listy', self.browser.title)
AssertionError: 'Listy' not found in 'Welcome to Django'

 ----------------------------------------------------------------------
Ran 1 test in 1.747s

FAILED (failures=1)
```

[1] Jedynym wyjątkiem jest sytuacja, gdy w metodzie `setUp()` nastąpi zgłoszenie wyjątku. W takim przypadku metoda `tearDown()` nie zostanie wykonana.

[2] *https://docs.python.org/3/library/unittest.html*

Otrzymane dane wyjściowe są teraz znacznie czytelniejsze. Okno przeglądarki internetowej Firefox zostało zamknięte, a na ekranie jest wyświetlany komunikat informujący o liczbie wykonanych testów i liczbie testów zakończonych niepowodzeniem. Ponadto `assertIn` wyświetla jasny komunikat błędu. Wspaniale!

Ukryte oczekiwanie

Na obecnym etapie mamy do wykonania jeszcze jedno zadanie: dodanie wywołania `implicitly_wait()` do metody `setUp()`:

```
    [...]
    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)

    def tearDown(self):
    [...]
```

To jest standardowe podejście stosowane w testach Selenium. Wprawdzie narzędzie Selenium całkiem dobrze radzi sobie z oczekiwaniem na pełne wczytanie strony, zanim spróbuje przeprowadzić na niej jakiekolwiek działania, ale jednak nie jest idealne na tym polu. Metoda `implicitly_wait()` nakazuje oczekiwanie przez podaną liczbę sekund, jeśli zachodzi potrzeba. W omawianym przykładzie narzędzie Selenium będzie czekało przez maksymalnie trzy sekundy, zanim rozpocznie wyszukiwanie tekstu na stronie.

Nie polegaj jedynie na metodzie `implicitly_wait()`, ponieważ nie sprawdza się ona w każdej sytuacji. Doskonale spełnia swoje zadanie w przypadku prostych aplikacji. Jak się jednak przekonasz w wyniku lektury części III książki (na przykład rozdziałów 15. i 20.), gdy poziom skomplikowana aplikacji przekroczy pewną wartość, wówczas konieczne jest opracowanie bardziej złożonych algorytmów *jawnego* oczekiwania przeznaczonych do użycia w testach.

Przekazanie plików do repozytorium

Teraz jest doskonały moment na umieszczenie kodu w repozytorium, ponieważ wprowadziliśmy pewną odizolowaną zmianę. Nasz test funkcjonalny został rozbudowany o komentarze opisujące zadania, które mamy do wykonania, aby powstała minimalna działająca wersja aplikacji w postaci listy rzeczy do zrobienia. Ponadto zmodyfikowaliśmy test, przystosowując go do użycia oferowanego przez Python modułu `unittest` oraz jego różnych funkcji pomocniczych z zakresu testów.

Wydaj polecenie `git status`, a przekonasz się, że jedynym zmodyfikowanym plikiem jest *functional_tests.py*. Następnie wydaj polecenie `git diff`, które pokaże różnice między projektem na dysku lokalnym a ostatnio umieszczonym w repozytorium. Dane wyjściowe drugiego z wymienionych poleceń powinny wskazać, że plik *functional_tests.py* uległ dość znacznym modyfikacjom:

```
$ git diff
diff --git a/functional_tests.py b/functional_tests.py
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
```

```
@@ -1,6 +1,45 @@
 from selenium import webdriver
+import unittest

-browser = webdriver.Firefox()
-browser.get('http://localhost:8000')
+class NewVisitorTest(unittest.TestCase):

-assert 'Django' in browser.title
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+        self.browser.implicitly_wait(3)
+
+    def tearDown(self):
+        self.browser.quit()
[...]
```

Teraz wydaj następujące polecenie:

```
$ git commit -a
```

Opcja `-a` oznacza „automatycznie dodaj wszelkie zmiany wprowadzone w monitorowanych plikach", czyli wszystkich plikach umieszczonych wcześniej w repozytorium. Powyższe polecenie nie spowoduje dodania do repozytorium nowych plików (w takim przypadku trzeba wyraźnie wydać polecenie `git add`). Jednak często — jak w omawianym przykładzie — nie ma żadnych nowych plików, więc wymienione polecenie stanowi użyteczny skrót.

Po wyświetleniu okna edytora tekstów dodaj opisowy komunikat dotyczący przekazywanych plików, na przykład „W komentarzach podano pierwszą specyfikację testu funkcjonalnego, a ponadto przystosowano go do użycia modułu unittest".

To jest doskonały moment na rozpoczęcie tworzenia faktycznego kodu naszej aplikacji w postaci listy rzeczy do zrobienia. Kontynuuj więc lekturę!

Użyteczne koncepcje TDD

Informacje od użytkownika
: Opis przedstawiający sposób działania aplikacji z punktu widzenia użytkownika. Ten opis służy do nadania struktury testowi funkcjonalnemu.

Oczekiwane niepowodzenie
: Test kończący się niepowodzeniem w oczekiwany sposób.

ROZDZIAŁ 3.

Testowanie prostej strony głównej za pomocą testów jednostkowych

Na końcu poprzedniego rozdziału przygotowaliśmy test funkcjonalny, którego wykonanie kończy się niepowodzeniem. Celem testu jest sprawdzenie, czy strona główna zawiera w tytule słowo *Listy*. Najwyższy czas rozpocząć prace nad właściwą aplikacją.

Ostrzeżenie: zaczynamy działać na poważnie

Pierwsze dwa rozdziały celowo były dość proste. Jednak od tej chwili poważnie bierzemy się za tworzenie kodu. Możemy przewidzieć jedno: w pewnym momencie coś na pewno pójdzie nie tak. Istnieje więc prawdopodobieństwo, że otrzymane przez Ciebie wyniki będą inne niż przedstawione w książce. To jest dobre, ponieważ stanowi niezwykle ważny element nauki.

Jednym z powodów mogą być niejasne wyjaśnienia z mojej strony, które spowodują, że zrobisz zupełnie co innego, niż miałem na myśli. W takim przypadku zrób krok wstecz i zastanów się, co chciałeś osiągnąć w danym miejscu. Zadaj sobie kilka pytań, na przykład: które pliki były edytowane, do czego chcesz skłonić użytkownika aplikacji, co jest testowane i dlaczego? Być może przeprowadziłeś edycję niewłaściwego pliku bądź funkcji lub też wykonałeś inne testy, niż powinieneś. Jestem przekonany, że w tego rodzaju sytuacjach i momentach zastanowienia możesz dowiedzieć się więcej o technikach TDD niż w pozostałych miejscach, gdy po prostu wykonujesz kolejne wskazywane polecenia.

Nie można również wykluczyć powstania faktycznego błędu w kodzie i wówczas musisz być nieustępliwy. Dokładnie przeczytaj cały komunikat błędu (patrz przedstawiona w dalszej części rozdziału ramka dotycząca stosu wywołań) od początku do końca. Najczęstsze przyczyny błędów to brak przecinka czy ukośnika lub litera s w nazwie jednej z metod narzędzia Selenium. Jak to trafnie ujął Zed Shaw[1], tego rodzaju usuwanie błędów stanowi absolutnie fantastyczną część nauki, więc działaj wytrwale.

Jeżeli wypróbowałeś różne możliwości i nadal nie znajdujesz rozwiązania, zawsze możesz wysłać do mnie wiadomość e-mail (lub zajrzeć na *forum poświęcone książce*[2]). Powodzenia podczas usuwania błędów!

[1] Zed A. Shaw, *Learn Python The Hard Way*: *http://learnpythonthehardway.org/*

[2] *https://groups.google.com/forum/#!forum/obey-the-testing-goat-book*