はじめての Django アプリ作成、その 5

このチュートリアルは チュートリアル 4 の続きです。Web 投票アプリケーションが完成したので、今度は自動テストを作ってみましょう。

> **困ったときは:**
>
> このチュートリアルの実行に問題がある場合は、FAQ の Getting Help セクションに進んでください。

自動テストの導入

自動テストとは何ですか？

Tests are routines that check the operation of your code.

テストは異なるレベルで実行されます。あるテストは、小さな機能に対して行われるもの (ある特定のモデルのメソッドは期待通りの値を返すか？) かもしれませんし、別のテストは、ソフトウェア全体の動作に対して行われるもの (サイト上でのユーザの一連の入力に対して、期待通りの結果が表示されるか？) かもしれません。こうしたテストは、前に チュートリアル その2 で **shell** を用いてメソッドの動作を確かめたことや、実際にアプリケーションを実行して値を入力して結果がどうなるのかを確かめるといったことと、何も違いはありません。

自動 テストが他と異なる点は、テスト作業がシステムによって実行されることです。一度テストセットを作成すると、それからはアプリに変更を加えるたびに、あなたの意図した通りにコードが動作するか確認できます。手動でテストする時間がかかることはありません。

なぜテストを作成する必要があるのか

どうしてテストを作るのか？また、なぜ今なのか？

もしかしたら、 Python や Django を学ぶのに手一杯で、さらに別のことを学ぶのは大変で不必要なことだと思われるかもしれません。だって投票アプリケーションはきちんと動いているし、わざわざ自動テストを導入したところでアプリケーションがより良くなるわけではないのだから。もし Django プログラミングを学ぶ目的がこの投票アプリケーションを作ることだけならば、確かに自動テストの導入は必要ないと思います。しかし、そうではないなら、今こそ自動テストについて学ぶ絶好の機会です。

テストはあなたの時間を節約します

ある程度のところまでは、'（動かしてみて、）正しく動いていそうであることを確認する' だけでもテストとして十分でしょう。高機能なアプリケーションでは、コンポーネント間の複雑な相互作用が数多くあるかもしれません。

それらのコンポーネントのどれかを変更した場合、予想外の振る舞いをアプリケーションがする可能性があります。'正しく動いているらしい' メソッドを使う場合、プログラムを壊していないことを確かめるためには、様々なテストデータを用いてプログラムを走らせる必要があります。これは、良い時間の使い方ではありません。

自動テストを導入することによってプログラムが正しく動くことの確認を一瞬で終わらせることができ、またテストはプログラムのどこで予期せぬ動作が起きたかを見極めるのに役立つことでしょう。

テストを書くという行為は、特にプログラムが適切に動くと分かっているときには、生産的でも創造的でもないつまらないことのように思われるかもしれません。

しかしテストを書くことは、何時間もかけてアプリケーションの動作を確認したり、新しく発生した問題の原因を探したりすることよりもずっとやりがいのあることなのです。

またテストは問題点を検出するのみならず、問題が発生するのを防ぐこともできます。

テストを単に開発の負の面と考えることは誤りです。

テストなくしては、アプリケーションの目的や意図した動作というものが曖昧になってしまうことがあります。自分自身で書いたコードであっても、時にはそのコードがすることを正確に理解するのに時間がかかってしまうことがあります。

テストはこの状況を大きく変え、いわばコードを内側から照らし出してくれます。そして、何か間違ったことをしてしまった時には、自分自身では間違っていると気づかなかった場合でさえ、間違いが起きた場所にスポットライトを当ててくれるのです。

テストは、コードをより魅力的にします

たとえあなたが輝かしいソフトウェアを作ったとしても、テストがないというそれだけの理由で、多くの開発者は見ることさえしてくれないでしょう。テストのないソフトは信用されないのです。Django を開発した Jacob Kaplan-Moss は次の言葉を残しています。「テストのないコードは、デザインとして壊れている。」

あなたのソフトウェアを他の開発者が真剣に見てもらうというのも、テストを書くべきもう一つの理由です。

テストを書くことはチームで共同作業を行う上で役に立ちます。

これまでの点は、1人の開発者でアプリケーションをメンテナンスしているという観点から書きました。しかし、複雑なアプリケーションはチームでメンテナンスされるようになるものです。テストは、あなたが書いたコードを他人がうっかり壊してしまうことから守ってくれます (そして、他の人が書いたコードをあなたが壊してしまうことからも)。Django のプログラマとして生きてゆくつもりなら、良いテストを絶対に書かなければなりません！

基本的なテスト方針

テストを書くためのアプローチには、さまざまなものがあります。

プログラマの中には、「テスト駆動開発」の原則に従っている人がいます。これは、実際にコードを書く前にテストを書く、という原則です。この原則は直感に反するように感じるかもしれませんが、実際には多くの人がどんなことでも普通にしていることに似ています。つまり、問題をきちんと言葉にしてから、その問題を解決するためのコードを書く、ということです。テスト駆動開発は、ここで言う問題を単に Python のテストケースとして形式化しただけのことです。

テストの初心者の多くは、先にコードを書いてから、その後でテストが必要だと考えるものです。おそらく、もっと早くからいくつかテストを書いておいた方が良いですが、テストを始めるのに遅すぎるということはありません。

どこからテストを書き始めるべきか、把握が難しい場合もあります。もしすでに数千行の Python コードがあったとしたら、テストすべき場所を選ぶのは簡単ではないかもしれません。そのような場合には、次に新しい機能の追加やバグの修正を行う時に、最初のテストを書いてみると役に立つでしょう。

それでは早速始めてみましょう。

初めてのテスト作成

バグを見つけたとき

運よく、 **polls** のアプリケーションにはすぐに修正可能な小さなバグがありました。**Question.was_published_recently()** のメソッドは **Question**が昨日以降に作成された場合に **True** を返すのですが(適切な動作)、 **Question** の **pub_date** が未来の日付になっている場合にも **True** を返してしまいます(不適切な動作)。

未来の日付の質問のメソッドをチェックするには、 **shell** を使用してバグを確認してください。

□/□ □

```
$ python manage.py shell
```

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30
days in the future
>>> future_question =
Question(pub_date=timezone.now() +
datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

未来の日付は '最近' ではないため、この結果は明らかに間違っています。

バグをあぶり出すためにテストを作成する

問題をテストするために **shell** でたった今したことこそ、自動テストでしたいことです。そこで、今やったことを自動テストに変換してみましょう。

アプリケーションのテストを書く場所は、慣習として、アプリケーションの **tests.py** ファイル内ということになっています。テストシステムが **test** で始まる名前のファイルの中から、自動的にテストを見つけてくれます。

polls アプリケーションの **tests.py** ファイルに次のコードを書きます。

polls/tests.py

```
import datetime

from django.test import TestCase
from django.utils import timezone

from .models import Question


class QuestionModelTests(TestCase):

    def
test_was_published_recently_with_future_question(self
):
        """
        was_published_recently() returns False for
questions whose pub_date
        is in the future.
        """
        time = timezone.now() +
datetime.timedelta(days=30)
        future_question = Question(pub_date=time)

self.assertIs(future_question.was_published_recently(
), False)
```

ここでは、未来の日付の **pub_date** を持つ **Question** のインスタンスを生成するメソッドを持つ **django.test.TestCase** を継承したサブクラスを作っています。それから、**was_published_recently()** の出力をチェックしています。これは False になるはずです。

テストの実行

ターミナルから、次のコマンドでテストが実行できます。

□/□ □

```
$ python manage.py test polls
```

すると、次のような結果になるでしょう。

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
======================================================================
FAIL:
test_was_published_recently_with_future_question
(polls.tests.QuestionModelTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question

self.assertIs(future_question.was_published_recently(
), False)
AssertionError: True is not False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Different error?

代わりにここで **NameError** を取得している場合、Part 2 のステップを見逃している可能性があります。そのステップでは、**polls/models.py** に **datetime** と **timezone** のインポートを追加しました。そのセクションからインポートを複製し、テストを再度実行してください。

ここでは以下のようなことが起こりました。

- **manage.py test polls** は、**polls** アプリケーション内にあるテストを探します
- **django.test.TestCase** クラスのサブクラスを発見します
- テストのための特別なデータベースを作成します
- テスト用のメソッドとして、**test** で始まるメソッドを探します
- **test_was_published_recently_with_future_question** の中で、**pub_date** フィールドに今日から30日後の日付を持つ **Question** インスタンスが作成されます
- そして最後に、 **assertIs()** メソッドを使うことで、本当に返してほしいのは **False** だったにもかかわらず、 **was_published_recently()** が **True** を返していることを発見します

テストは私たちにテストの失敗を教えてくれるだけでなく、失敗が起こったコードの行数まで教えてくれています。

バグを修正する

私たちはすでに問題の原因を知っています。それは、**Question.was_published_recently()** は **pub_date** が未来の日付だった場合には **False** を返さなければならない、ということです。**models.py** にあるメソッドを修正して、日付が過去だった場合にのみ **True** を返すようにしましょう。

polls/models.py

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <=
self.pub_date <= now
```

そして、もう一度テストを実行します。

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

バグを発見した後、私たちはそのバグをあぶり出してくれるようなテストを書いて、コード内のバグを直したので、テストは無事にパスされました。

このアプリケーションでは将来、たくさんの他のバグが生じるかもしれませんが、このバグがうっかり入ってしまうことは二度とありません。単にテストを実行するだけで、すぐに警告を受けられるからです。アプリケーションのこの小さな部分が、安全に、そして永遠にピン留めされたと考えて差し支えありません。

より包括的なテスト

この段階で、**was_published_recently()** メソッドをさらにピン留めしておけます。実際、一つのバグを直したことでほかのバグを作り出すなんてしたくありませんよね。

このメソッドの振る舞いをより包括的にテストするために、同じクラスにさらに2つのテストを追加しましょう。

polls/tests.py

```
def
test_was_published_recently_with_old_question(self):
    """
    was_published_recently() returns False for
questions whose pub_date
    is older than 1 day.
    """
    time = timezone.now() -
datetime.timedelta(days=1, seconds=1)
    old_question = Question(pub_date=time)

self.assertIs(old_question.was_published_recently(),
False)

def
test_was_published_recently_with_recent_question(self
):
    """
    was_published_recently() returns True for
questions whose pub_date
    is within the last day.
    """
    time = timezone.now() -
datetime.timedelta(hours=23, minutes=59, seconds=59)
    recent_question = Question(pub_date=time)

self.assertIs(recent_question.was_published_recently(
), True)
```

これで、**Question.was_published_recently()** が過去、現在、そして未来の質問に対して意味のある値を返すことを確認する3つのテストが揃いました。

先述したように **polls** は簡単なアプリケーションですが、メソッドに対してテストを書いたおかげで、将来このアプリケーションがどんなに複雑になっても、あるいは他のどんなコードと相互作用するようになっても、メソッドが期待どおりに動作することを保証できるようになったのです。

ビューをテストする

この投票アプリケーションは、まだ質問をちゃんと見分けることができません。**pub_date** フィールドが未来の日付になっている質問を含め、どんな質問でも公開してしまいます。この点を改善するべきでしょう。**pub_date** を未来に設定するということは、その Question がその日付になった時に公開され、それまでは表示されないことを意味するはずです。

ビューに対するテスト

上でバグを修正した時には、初めにテストを書いてからコードを修正しました。実は、テスト駆動開発の簡単な例だったわけです。しかし、テストとコードを書く順番はどちらでも構いません。

初めのテストでは、コード内部の細かい動作に焦点を当てましたが、このテストでは、ユーザが Web ブラウザを通して経験する動作をチェックしましょう。

初めに何かを修正する前に、使用できるツールについて見ておきましょう。

Django テストクライアント

Django は、ビューレベルでのユーザとのインタラクションをシミュレートすることができる **Client** を用意しています。これを **tests.py** の中や **shell**でも使うことができます。

もう一度 **shell** からはじめましょう。ここでテストクライアントを使う場合には、**tests.py** では必要がない2つの準備が必要になります。まず最初にしなければならないのは、**shell** の上でテスト環境をセットアップすることです。

□/□ □

```
$ python manage.py shell
```

```
>>> from django.test.utils import
setup_test_environment
>>> setup_test_environment()
```

setup_test_environment() は、テンプレートのレンダラーをインストールします。 これによって、今までは調査できなかった、レスポンス上のいくつかの属性（たとえば **response.context** ）を調査できるようになります。 注意点として、このメソッドはテスト用データベースを作成しないので、これに続く命令は既存のデータベースに対して実行されます。あなたの作成した question によってはアウトプットが多少異なるかもしれません。また **settings.py** の **TIME_ZONE** が正しくない場合は、予期しない結果になるでしょう。設定が正しいかどうか自信がなければ、次に進む前に TIME_ZONE を確認してください。

つぎに、テストクライアントのクラスをインポートする必要があります (後に取り上げる **tests.py** の中では、 **django.test.TestCase** クラス自体がクライアントを持っているため、インポートは不要です)。

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

さて、これでクライアントに仕事を頼む準備ができました。

```
>>> # get a response from '/'
>>> response = client.get('/')
Not Found: /
>>> # we should expect a 404 from that address; if
you instead see an
>>> # "Invalid HTTP_HOST header" error and a 400
response, you probably
>>> # omitted the setup_test_environment() call
described earlier.
>>> response.status_code
404
>>> # on the other hand we should expect to find
something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded
URL
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n    <ul>\n    \n        <li><a
href="/polls/1/">What&#x27;s up?</a></li>\n    \n
</ul>\n\n'
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>]>
```

ビューを改良する

現在の投票のリストは、まだ公開されていない (つまり **pub_date** の日付が未来になっている) 投票が表示される状態になっています。これを直しましょう。

Tutorial 4 では、以下のような **ListView**: をベースにしたクラスベースビューを導入しました。

polls/views.py

```
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published
questions."""
        return Question.objects.order_by('-pub_date')
[:5]
```

get_queryset() メソッドを修正して、日付を **timezone.now()** と比較してチェックする必要があります。まず、インポート文を追加します:

polls/views.py

```
from django.utils import timezone
```

そして、次のように **get_queryset** メソッドを修正します。

polls/views.py

```
def get_queryset(self):
    """
    Return the last five published questions (not
including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

Question.objects.filter(pub_date__lte=timezone.now()) は、**pub_date** が **timezone.now** 以前の **Question** を含んだクエリセットを返します。

新しいビューをテストする

それでは、これで期待通りの満足のいく動作をしてくれるかどうか確かめましょう。まず、 **runserver** を実行して、ブラウザでサイトを読み込みます。過去と未来、それぞれの日付を持つ **Question** を作成し、すでに公開されている質問だけがリストに表示されるかどうかを確認します。あなたはまさか、この通りにちゃんと動作しているか、プロジェクトにわずかでも変更を加えるたびに毎回手動で 確認したいなどとは思わないですよね？ それなら、今回も上の **shell** のセッションに基づいてテストを作りましょう。

まず、**polls/tests.py** に次の行を追加します。

polls/tests.py

```
from django.urls import reverse
```

そして、question を簡単に作れるようにするショートカット関数と、新しいテストクラスを作ります。

polls/tests.py

```
def create_question(question_text, days):
    """
    Create a question with the given `question_text`
and published the
    given number of `days` offset to now (negative
for questions published
    in the past, positive for questions that have yet
to be published).
    """
    time = timezone.now() +
datetime.timedelta(days=days)
    return
Question.objects.create(question_text=question_text,
pub_date=time)


class QuestionIndexViewTests(TestCase):
    def test_no_questions(self):
        """
        If no questions exist, an appropriate message
is displayed.
        """
        response =
self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are
available.")

self.assertQuerysetEqual(response.context['latest_que
stion_list'], [])

    def test_past_question(self):
        """
        Questions with a pub_date in the past are
displayed on the
        index page.
        """
        create_question(question_text="Past
```

```
question.", days=-30)
        response =
self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_future_question(self):
        """
        Questions with a pub_date in the future
aren't displayed on
        the index page.
        """
        create_question(question_text="Future
question.", days=30)
        response =
self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are
available.")

self.assertQuerysetEqual(response.context['latest_que
stion_list'], [])

    def test_future_question_and_past_question(self):
        """
        Even if both past and future questions exist,
only past questions
        are displayed.
        """
        create_question(question_text="Past
question.", days=-30)
        create_question(question_text="Future
question.", days=30)
        response =
self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
```

```
            ['<Question: Past question.>']
        )

    def test_two_past_questions(self):
        """
        The questions index page may display multiple
questions.
        """
        create_question(question_text="Past question
1.", days=-30)
        create_question(question_text="Past question
2.", days=-5)
        response =
self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>',
'<Question: Past question 1.>']
        )
```

これらのコードを詳しく見ていきましょう。

まず、question のショートカット関数 **create_question** です。この関数によって、 question 作成処理のコード重複をなくしています。

test_index_view_with_no_questions は question を1つも作りませんが、 "No polls are available." というメッセージが表示されていることをチェックし、**latest_question_list** が空になっているか確認しています。 **django.test.TestCase** クラスが追加のアサーションメソッドを提供していることに注意してください。この例では、 **assertContains()** と **assertQuerysetEqual()** を使用しています。

test_index_view_with_a_past_question では、question を作成し、その question がリストに現れるかどうかを検証しています。

test_index_view_with_a_future_question では、**pub_date** が未来の日付の質問を作っています。データベースは各テストメソッドごとにリセットされるので、この時にはデータベースには最初の質問は残っていません。そのため、index ページにはquestion は1つもありません。

以下のテストメソッドも同様です。実際のところ、私たちはテストを用いて、管理者の入力とサイトでのユーザの体験についてのストーリを語り、システムの各状態とそこでの新しい変化のそれぞれに対して、期待通りの結果が公開されているかどうかをチェックしているのです。

DetailView のテスト

とても上手くいっていますね。しかし、未来の質問は *index* に表示されないものの、正しいURL を知っていたり推測したりしたユーザは、まだページに到達できてしまいます。そのため、同じような制約を **DetailView** にも追加する必要があります。

polls/views.py

```
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published
yet.
        """
        return
Question.objects.filter(pub_date__lte=timezone.now())
```

We should then add some tests, to check that a **`Question`** whose **`pub_date`** is in the past can be displayed, and that one with a **`pub_date`** in the future is not:

`polls/tests.py`

```
class QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date
in the future
        returns a 404 not found.
        """
        future_question =
create_question(question_text='Future question.',
days=5)
        url = reverse('polls:detail', args=
(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_past_question(self):
        """
        The detail view of a question with a pub_date
in the past
        displays the question's text.
        """
        past_question =
create_question(question_text='Past Question.',
days=-5)
        url = reverse('polls:detail', args=
(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response,
past_question.question_text)
```

さらなるテストについて考える

ResultsView にも同じように **get_queryset** メソッドを追加して、新しいテストクラスも作らなければならないようです。しかしこれは、今作ったばかりのものとそっくりになるでしょう。実際、テストは重複だらけになるはずです。

テストを追加することによって、同じように他の方法でアプリを改善できるでしょう。例えば、**Choices** を一つも持たない馬鹿げた **Questions** が公開可能になっています。このような **Questions** を排除するようビューでチェックできます。 **Choices** がない **Question** を作成し、それが公開されないことをテストし、同じようにして、**Choices** がある **Question** を作成し、それが公開 される ことをテストすることになるでしょう。

もしかすると管理者としてログインしているユーザーは、一般の訪問者と違い、 まだ公開されていない **Questions** を見ることができるようにした方がいいかもしれません。また繰り返しになりますが、この問題を解決するためにソフトウェアにどんなコードが追加されべきであったとしても、そのコードにはテストが伴うべきです。テストを先に書いてからそのテストを通るコードを書くのか、あるいはコードの中で先にロジックを試してからテストを書いてそれを検証するのか、いずれにしてもです。

ある時点で、書いたテストが限界に達しているように見え、テストが膨らみすぎてコードが苦しくなってしまうのではないかという疑問が浮かんでくるでしょう。こうなった場合にはどうすれば良いのでしょうか？

テストにおいて、多いことはいいことだ

私たちのテストは、手がつけられないほど成長してしまっているように見えるかもしれません。この割合で行けば、テストコードがアプリケーションのコードよりもすぐに大きくなってしまうでしょう。そして繰り返しは、残りの私たちのコードのエレガントな簡潔さに比べて、美しくありません。

構いません。テストコードが大きくなるのに任せましょう。たいていの場合、あなたはテストを一回書いたら、そのことを忘れて大丈夫です。プログラムを開発し続ける限りずっと、そのテストは便利に機能し続けます。

時には、テストのアップデートが必要になることがあります。たとえば、 **Choices** を持つ **Questions** だけを公開するようにビューを修正したとします。この場合、既存のテストの多くは失敗します。この失敗によって、 最新の状態に対応するために、どのテストを修正する必要があるのか が正確にわかります。そのため、ある程度、テストはテスト自身をチェックする助けになります。

最悪の場合、開発を続けていくにつれて、あるテストが今では冗長なものになっていることに気づいてしまうかもしれません。これも問題ではありません。テストにおいては、冗長であることは 良い ことなのです。

きちんと考えてテストを整理していれば、テストが手に負えなくなることはありません。経験上、良いルールとして次のようなものが挙げられます。

- モデルやビューごとに **TestClass** を分割する
- テストしたい条件の集まりのそれぞれに対して、異なるテストメソッドを作る
- テストメソッドの名前は、その機能を説明するようなものにする

さらなるテスト

このチュートリアルでは、テストの基本の一部を紹介しました。この他にもあなたにできることはまだまだたくさんありますし、いろいろと賢いことを実現するに使えるとても便利なツールが数多く用意されています。

たとえば、ここでのテストでは、モデルの内部ロジックと、ビューの情報の公開の仕方をカバーしましたが、ブラウザが HTML を実際にどのようにレンダリングのするのかをテストする Selenium のような "in-browser" のフレームワークを使うこともできます。これらのツールは、Django が生成したコードの振る舞いだけでなく、たとえば、 JavaScript の振る舞いも確認できます。テストがブラウザを起動してサイトとインタラクションしているのを見るのはとても面白いですよ。まるで本物の人間がブラウザを操作しているかのように見えるんです！ Django には、Selenium のようなツールとの連携を容易にしてくれる **LiveServerTestCase** が用意されています。

複雑なアプリケーションを開発する時には、継続的インテグレーション (continuous integration) のために、コミットの度に自動的にテストを実行するといいかもしれませんね。継続的インテグレーションを行えば、品質管理それ自体が、少なくとも部分的には自動化できます。

アプリケーションのテストされていない部分を発見するには、コードカバレッジをチェックするのが良いやり方です。これはまた、壊れやすいコードや使用されていないデッドコードの発見にも役に立ちます。テストできないコード片がある場合、ふつうは、そのコードはリファクタリングするか削除する必要があることを意味します。カバレッジはデッドコードの識別に役に立つでしょう。詳細は Integration with coverage.py を参照してください。

Django におけるテスト には、テストに関する包括的な情報がまとめられています。
