## はじめに

Pythonで開発されたWebアプリケーションフレームワークであるDjango（ジャンゴと読む）はPythonの簡潔さをうまく生かし、シンプルかつ本格的な開発ができるWebアプリケーションフレームワークです。**本稿の前編**では、DjangoやPythonの特徴、Djangoの概要の紹介とインストール手順、そして、チュートリアルとして、商品（Item）のコマンドラインシェルからのデータ操作、そして、ビュー関数によるHTMLへの表示までを説明しました。後編となる今回は、管理インターフェイスの使い方、ショートカット、汎用ビュー、 Form クラス、セッション、キャッシュの使用方法などについて説明します。

## 対象読者

- PythonによるWebアプリケーション開発に興味がある方
- 日ごろ、Perl、Ruby、PHP、Java、C#などPython以外のプログラミング言語で開発している方
- Webアプリケーションの開発をこれから学ぶ方、もしくは学びはじめたばかりの方。

## 必要な環境

- Windows XP
- Python2.6.2
- Django1.1
- SQLite 3

## 管理インターフェイス

管理インターフェイスは、モデル定義をもとにそのモデルのCRUD（Create、Read、Update、Delete）用の管理画面を提供する機能です。Webサービスでは一般に、そのサービスで提供される表向きの機能とは別に、データを管理するためのバックエンド（バックオフィス）用の定型的な画面が必要となりますが、管理インターフェイスはこのバックエンド用画面を簡単に提供することを目的としています。ここでは、前編で作成した Item クラスを管理インターフェイスで操作する方法を順番に説明します。

### 管理インターフェイスの有効化

管理インターフェイスはデフォルトでは無効化されているので、有効化する必要があります。管理インターフェイスの有効化の手順は以下とおりです。

1. 管理インターフェイス用のアプリケーションをプロジェクトに追加（settings.pyのINSTALLED_APPS変数に追加）
2. 管理インターフェイス用テーブルをDBに同期
3. 管理インターフェイス用のURL定義をURLディスパッチャ（urls.py）に追加
4. ブラウザでアクセス

#### settings.pyの編集

管理インターフェイスのアプリケーションを有効化するには、管理インターフェイスのアプリケーションパッケージである「django.contrib.admin」をプロジェクトフォルダのsettings.pyのINSTALLED_APPS変数に追加します。

［リスト1］settings.pyの編集（管理インターフェイスのアプリケーション追加）

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'ecsite.itempage',
    'django.contrib.admin',
)
```

#### DBと管理インターフェイス用モデルの同期

次に管理インターフェイスのモデルとDBを同期させます。プロジェクトフォルダで、以下のコマンドを実行します。

［リスト2］管理インターフェイスのモデルをDBへ同期

```
C:¥codezine¥ecsite> manage.py syncdb
```

▼

```
Creating table django_admin_log
Installing index for admin.LogEntry model
```

### urls.py の編集

次に「http://hostname:8000/admin」のURLで管理インターフェイス用のビュー関数が実行されるようにするため、プロジェクトフォルダのurls.pyにリスト3の記述を追加します。urls.pyには該当のソースが既にコメントで記載されているので、コメントを外します。

［リスト3］urls.pyの編集

```
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    ～省略
    (r'^admin/', include(admin.site.urls)),
)
```

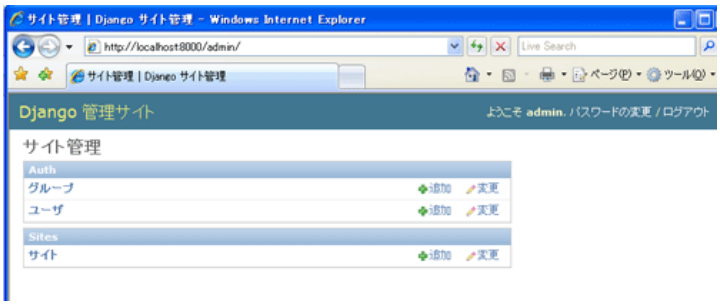
ブラウザで「http://localhost:8000/admin」にアクセスして動作を確認します。

図1 管理インターフェイスのログイン画面


ログイン画面が表示されたら、プロジェクト作成時にプロンプトから入力した、管理者用のID／パスワードを入力して（前編では、ID:admin、パスワード:adminを入力）ログインしてみましょう。

ログインに成功すると図2の画面が表示されます。

図2 管理インターフェイスのログイン後画面




### 管理インターフェイスにItemモデルを追加する

管理インターフェイスのログイン後画面では、まだitempageアプリケーションのモデルを操作するため機能は表示されていません。次にItemモデルを操作できるように設定してみましょう。

管理インターフェイスに、アプリケーション独自のデータを操作するための機能を追加するには、アプリケーションフォルダにadmin.pyというファイルを作成します。admin.pyに次のような記述をします。

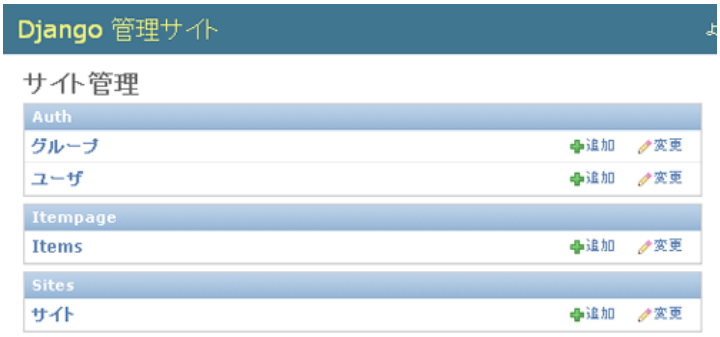
［リスト4］admin.pyの編集（管理インターフェイスへItemモデルを追加）

```
# -*- coding: utf-8 -*-
from django.contrib import admin
from ecsite.itempage.models import Item

admin.site.register(Item)
```

管理インターフェイスログイン後の画面でブラウザをリロードしてみましょう（開発サーバの再起動は必要ありません）。図3のようにItemのモデルを編集するためのリンクが表示されました。

図3 管理インターフェイスのログイン後画面




以上の操作だけで、Itemの一覧表示、新規登録、編集、削除などが実行できるようになります。Itemの一覧画面の「itemを追加」ボタンを押下すると、図4のItem新規登録画面が表示されます。

図4 管理インターフェイスのItem新規登録画面


一覧画面の一覧上のリンクをクリックすると、Itemの編集画面が表示されます。編集画面には、該当Itemの削除ボタンもあり、削除することができます。これでItemのデータがひととおり操作できることがお分かりいただけることと思います。

図5 管理インターフェイスのItem編集画面


## Itemモデル一覧のカスタマイズ

次に商品の一覧表示で商品コード以外にも、商品名、価格が表示されるようにカスタマイズしてみましょう。Djangoに用意されている一覧表示のデフォルト表示は、商品コード（item_code）だけが表示されています。これはモデルのUnicode 表記（Itemモデルの）が呼び出され表示されているものです。

［リスト5］models.py（Itemモデルの__unicode__関数）

```
class Item(models.Model):
    item_code = models.CharField(u'商品コード',max_length=256,unique=True)
    ～省略
    def __unicode__(self):
        return self.item_code
```

管理インターフェイス上のモデル操作をカスタマイズするには、アプリケーションフォルダのadmin.pyに ModelAdmin（django.contrib.admin）を継承したクラスを作成し、そこにカスタマイズ内容を定義します。ここでは ModelAdminを継承した ItemAdmin クラスを作成します。一覧表示に表示したい項目を指定するには、 list_display 属性に Pythonのタプル形式でモデルのフィールド名

（item_code、item_name、price）を定義します。クラスを定義後、 admin.site.register メソッドの引数に、ItemAdminも渡すように修正します。リスト6に修正後のadmin.pyを示します。

［リスト6］admin.py（ItemAdminクラスの追加）

```
# -*- coding: utf-8 -*-
from django.contrib import admin
from ecsite.itempage.models import Item

class ItemAdmin(admin.ModelAdmin):
    list_display = ('item_code', 'item_name','price') # 一覧表示したいフィールドを設定

admin.site.register(Item,ItemAdmin) #ItemAdminクラスも追加
```

admin.pyに定義したら開発サーバを再起動し、Itemモデルの一覧表示画面を表示してみましょう。図6のように商品コード（item_code）、商品名（item_name）、価格（price）が表示されます。

図6 Itemモデルの一覧画面


Django の管理インターフェイスをカスタマイズするには、今回紹介したようにadmin.pyファイルにカスタマイズ内容を定義していきます。今回紹介した以外にも、さまざまなカスタマイズが提供されていて柔軟にカスタマイズできることが管理インターフェイスの特徴です。それらの方法については**Django本家サイト**のドキュメントを参照してください。

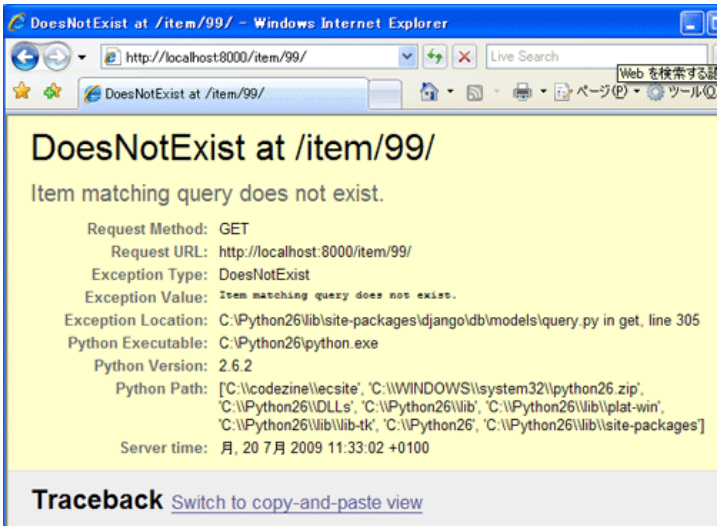
### Djangoの管理インターフェイスを用いたアプリケーションの例

Django の管理インターフェイスのうまくカスタマイズした事例としては**Review Board**があげられます。Review BoardはVM Wareの開発チームが使用しているDjangoで開発されたソースレビュー用のアプリケーションです。管理インターフェイスのカスタマイズ事例として参考にすると良いでしょう。

## HTTPステータスの制御

前編で、商品ページの表示機能を実装しましたが、存在しないIDのURLが指定された場合、図7のようにサーバーエラー画面（HTTPレスポンスコード500）が表示されてしまいます。

図7 Itemモデルが存在しない場合のサーバエラー




ここはサーバエラーにならずにHTTPレスポンスコードの404（ページが存在しない）を返すように実装してみましょう。モデルの取得に使用している「Item.objects.get」関数は、データが存在しない場合Item.DoesNotExist例外を送出しますのでexcept節でハンドリングし、404レスポンスコードを返すためのオブジェクトをraiseします。この404レスポンスコードを返すためのオブジェクトはdjangoが用意しているクラスである

django.http.Http404を使用します。アプリケーションフォルダのviews.pyの「item_page_display」ビュー関数を、リスト7のように編集します。

［リスト7］views.py（データが存在しない場合、404エラーを返す）

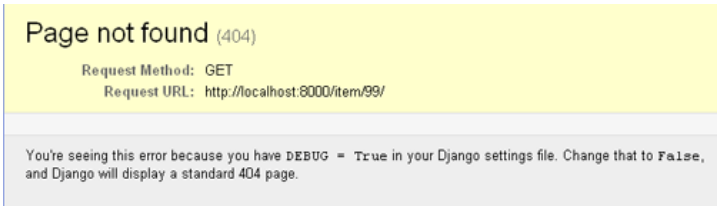
```
from django.http import Http404

def item_page_display(request,item_id):
    try:
        item = Item.objects.get(id=item_id)
    except Item.DoesNotExist:
        raise Http404

    t = loader.get_template('page/item.html')
    c = Context(
        {'item':item }
    )
    return HttpResponse(t.render(c))
```

記述を追加後、商品ページをリロードしてみましょう。図8のように404エラーが表示されることが確認できます。

図8 Itemモデルが存在しない場合の404エラー




## ショートカットの使用

ここでさらに一歩進めて「DBを検索してデータがなかったらHTTPレスポンスコード404を返す」という記述は定型的であり、誰が書いても大体同じ記述になるので共通化したいところです。Djangoではそのような定型的な記述を簡潔に記述できるようにするための「ショートカット（Shortcuts）」というライブラリを用意しています。ここでは「get_object_or_404」というショートカットを使用して、404エラーを返せるようにします。リスト8にショートカットを使用したソースを示します。

［リスト8］views.py（get_object_or_404ショートカットを使用して404エラーを返す）

```
from django.shortcuts import get_object_or_404

def item_page_display(request,item_id):

    item = get_object_or_404(Item,id=item_id)  #ショートカットを使用（データが存在しない場合404エラーが返る）

    t = loader.get_template('page/item.html')
    c = Context(
        {'item':item }
    )
    return HttpResponse(t.render(c))
```

ブラウザをリロードすると、図8と同じ404エラーが表示されます。

## 汎用ビュー（Generic Views）

ここでitem_page_displayビュー関数のコードをさらに短くしてみましょう。コードの後半部分で「Templateインスタンスを取得後、 Contextのインスタンスを生成し、Tempateの render メソッドでレンダリングしたHTML文字列をHttpResponseのコンストラクタに渡して返す」という処理が記述されていますが、これもビューに関する決まりきった処理であると言えます。このようにビューの処理に関する定型的な処理を何度も書かなくて済むように汎用化した機能が、Djangoでは「汎用ビュー（Generic Views）」として用意されています。

### direct_to_template汎用ビューの適用

ここではdirect_to_template汎用ビューを使用します。direct_to_template汎用ビューは引数に request オブジェクトと、リダイレクト先テンプレート名、そしてテンプレートにマージしたい値を格納した辞書（dictionary）を引数に指定します。 direct_to_template汎用ビューをitem_page_displayビュー関数で使用したのがリスト9です。なお汎用ビューでは共通の省略可能な引数を用意していて、デフォルト値から変更したい場合は必要に応じて引数を指定します。今回はその共通引数は指定しません。

［リスト9］views.py（汎用ビューのdirect_to_templateを使用）

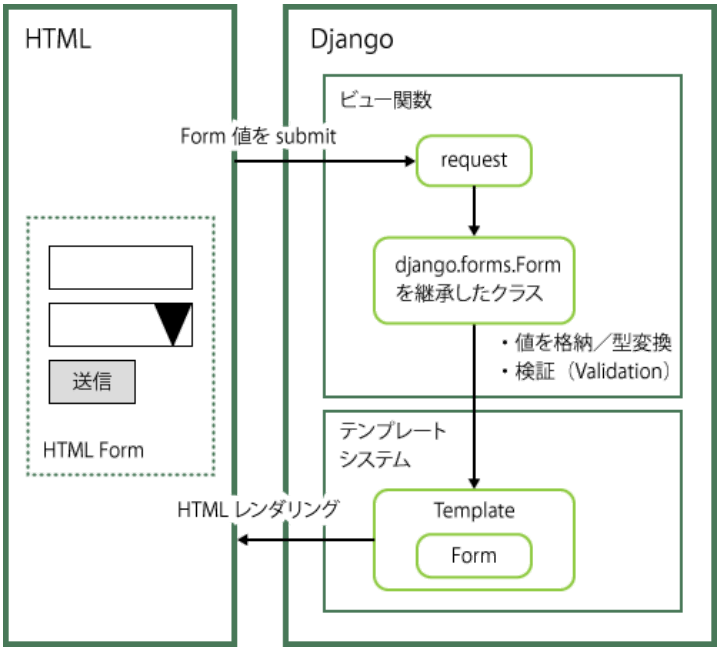
```
def item_page_display(request,item_id):
    item = get_object_or_404(Item,id=item_id)
```

```
    return direct_to_template(
        request,'page/item.html',
        extra_context={'item': item}
    )
```

## Formの使用

商品ページの表示、管理インターフェイスによる商品の登録・更新・削除機能と作成してきましたが、次は商品名で商品を検索して一覧表示する機能を作成します。Djangoでは、HTMLフォームの値を受け取り、バリデーション、値の型変換機能を提供するForm機能（django.forms）を用意しています。Form機能の使用イメージを図9に示しました。

図9 Form機能の使用イメージ




### Formクラスの作成

Form クラスの記述はアプリケーションフォルダにforms.pyというファイルを作成し記述します（リスト10）。

［リスト10］forms.py

```
# -*- coding: utf-8 -*-
from django import forms

class ItemSearchForm(forms.Form):
    item_name = forms.CharField()
```

### ビュー関数の記述

次に初期の入力フォームを表示する処理と、フォームに入力された値をもとに、商品を検索して一覧表示するためのitem_searchビュー関数をviews.pyに記述します。リスト11にソースを示しました。

［リスト11］views.py

```
def item_search(request):

    if request.method == 'POST':
        form = ItemSearchForm(request.POST)
        if form.is_valid():
            items = Item.objects.filter(item_name=form.cleaned_data['item_name'])
            return direct_to_template(request,'page/item_search.html',
                extra_context={
                'form': form,
                'items': items,
                }
            )
    else:
        #検索フォームの初期表示
        form = ItemSearchForm()

    return direct_to_template(request,'page/item_search.html',
        extra_context={
        'form': form,
    })
```

ビュー関数の処理は、HTTPのmethodがGETの場合とPOSTの場合で分かれます。GETの場合は検索フォームを初期表示します。ItemSearchFormの空のコンストラクタで生成したインスタンスをコンテキストに渡して、入力フィールドをレンダリングします。テキストフィールドのレンダリングなどは Form クラスの機能で実現されます。POSTの場合は、検索フォームで値が入力され、検索ボタンが押下された時の処理を実行します。入力された商品名を元に「 Item.objects.filter(item_name=form.cleaned_data['item_name']) 」とモデルでDBを検索し、Itemモデルのリストを取得します。取得したリストは、direct_to_templateに渡すことで、テンプレートにレンダリングします。

### 検索フォームのテンプレート作成

次に検索フォームと検索結果を表示するテンプレートを作成しましょう。リスト12にソースを示しました。

[リスト12] item_search.html

```
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
   <title>商品検索</title>
<head>
<body>
<center>
{{ error_message_no_item }}
<form action="/itemsearch" method="POST">
商品名を入力してください：{{ form.item_name }}<br><br>
<input type="submit" value="検索">
</form>
<br>
<table border="1">
   <th>商品コード</th><th>商品名</th><th>価格</th>
{% for item in items %}
   <tr><td>{{ item.item_code }}</td>
       <td>{{ item.item_name }}</td>
       <td>{{ item.price }}</td>
   </tr>
{% endfor %}
</table>

</body>
</html>
```

### URLマッピングの定義

最後に検索処理用のビュー関数と、URLをマッピングさせる処理をurls.pyに記述します（リスト13）。

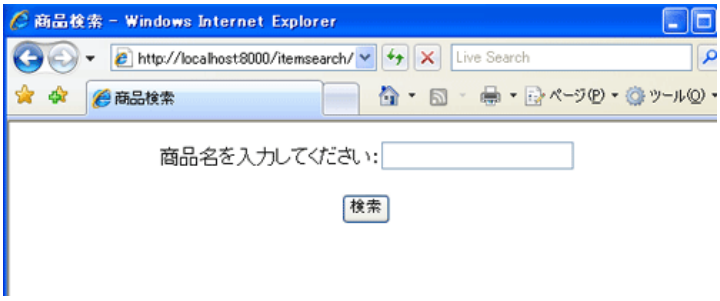
[リスト13] urls.py

```
urlpatterns = patterns('',
    (r'^admin/', include(admin.site.urls)),
    (r'^item/(?P<item_id>¥d+)/$', 'ecsite.itempage.views.item_page_display'),
    (r'^itemsearch', 'ecsite.itempage.views.item_search'),
)
```

### 実行

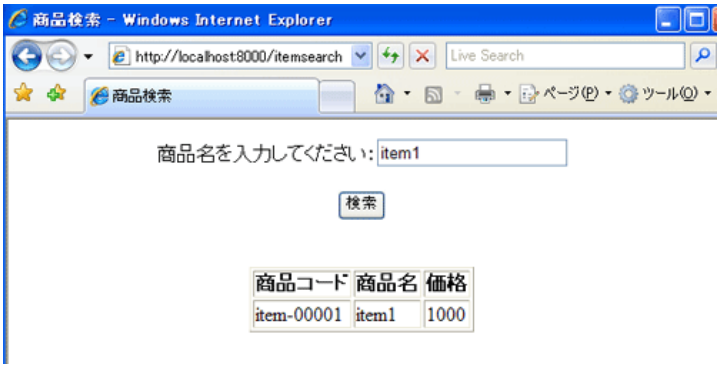
それでは商品検索処理を実行してみましょう。ブラウザで「http://localhost:8000/itemsearch」にアクセスし検索画面を表示します。

図10 商品検索フォーム




商品名入力欄に商品登録時に指定した商品名を入力して（本稿では「item1」）検索ボタンを押下します。図11のように一覧画面が表示されます。

図11 商品検索結果（商品一覧）




## HTTPセッションの使用

次にWebアプリケーションでは必須の機能となるセッション機能を使用してみましょう。ECサイトに買い物かご機能を付加して複数の商品を購入できるように機能を追加します。

### セッションの有効化

まずはDjangoのプロジェクトでセッション機能用のミドルウェア（django.contrib.sessions.SessionMiddleware）とアプリケーション（django.contrib.sessions）を有効化するための設定をsettings.py（リスト14）に記述する必要があります。この設定はデフォルトで有効になっていますので、特に編集する必要はありません。

［リスト14］settings.py

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
～省略
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    ～省略
)
```

### 買い物かごの実装

#### 商品ページの修正

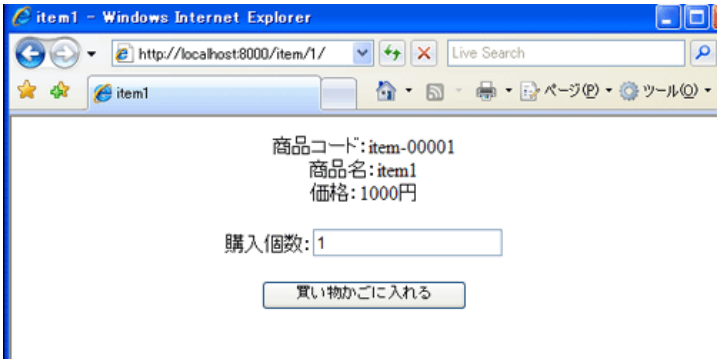
買い物かごの機能は商品ページの「買い物かごに入れる」ボタンを押下すると、セッション情報に、購入商品情報（商品、購入数、価格など）を格納し、買い物かごの中身を画面に表示するという機能とします。

まずは商品ページに買い物かごに遷移するための機能を追加するためテンプレートを修正します。リスト15のようにテンプレートを修正してください（**太字**の箇所を追加しています）。formを追加し、商品の購入個数とitem_id（hidden）を送信できるようにしています。図12が修正後の画面イメージです。

［リスト15］item.html（商品ページ）

```
<html>
～省略
商品コード：{{ item.item_code }}<br>
商品名：{{ item.item_name }}<br>
価格：{{ item.price }}円<br>
<br>
<form action="/cart" method="POST">
購入個数：<input type="text" name="buy_num" value="1"><br>
<br>
<input type="submit" value="買い物かごに入れる">
<input type="hidden" name="item_id" value="{{ item.id }}">
</form>
～省略
</html>
```

図12 商品ページへの購入個数入力欄追加




### 買い物かご用ビュー関数の実装

買い物かごの機能を実現するためのビュー関数は、アプリケーションフォルダのview.pyのdo_cartビュー関数に記述します。購入情報は、セッションにリスト形式で保存します。まずは、リストに格納する購入情報を格納するクラスをリスト16に示します。

［リスト16］cart.py（商品購入情報格納用クラス）

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

class CartItem:
    item_id = 0
    item_code = None
    item_name = None
    price = 0
    buy_num =0
```

次に、買い物かご情報を操作するビュー関数（do_cart）をリスト17に示します。

［リスト17］views.py（買い物かご制御用ビュー関数）

```
def do_cart(request):

    # hiddenのitem_idを取得しintに型変換
    item_id = int(request.POST['item_id'])
    item = Item.objects.get(id=item_id)

    # DBからItem情報を取得する。
    cart_item_list = request.session.get('cart_item_list',[])
    ci = CartItem()
    ci.item_id = item_id
    ci.item_code = item.item_code
    ci.item_name = item.item_name
    ci.price = item.price
    ci.buy_num = request.POST['buy_num']
    cart_item_list.append(ci)
    request.session['cart_item_list'] = cart_item_list

    return direct_to_template(request,'page/cart_item_list.html',
        extra_context={
            'cart_item_list': cart_item_list,
        }
    )
```

> **注**
>
> POSTされたデータを取得し、データ処理をする際には、前掲したFormクラスを使用するのが通常ですが、ここではサンプルの簡略化を図るため、request.POSTから取得した値をint変換する方法を採用しました。

### 買い物かご中身表示用テンプレートの定義

ビュー関数で制御されている買い物かご情報を表示するテンプレートがcart_item_list.htmlです。リスト18にテンプレートの内容を示します。

［リスト18］cart_item_list.html（買い物かご制御用ビュー関数）

```
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
   <title>買い物かご</title>
<head>
<body>
<center>買い物かごの中身は次のとおりです。<br>
<table border="1">
   <tr><th>商品コード</th><th>商品名</th><th>価格</th><th>購入個数</th></tr>
{% for cart_item in cart_item_list %}
   <tr>
      <td>{{ cart_item.item_code }}</td>
      <td>{{ cart_item.item_name }}</td>
```

```
      <td>{{ cart_item.price }}円</td>
      <td>{{ cart_item.buy_num }}個</td>
    </tr>
{% endfor %}
</table>
</center>
</body>
</html>
```

### urls.pyの定義

最後にdo_cartビュー関数とURLをマッピングをurls.pyファイルに設定します（リスト19）。

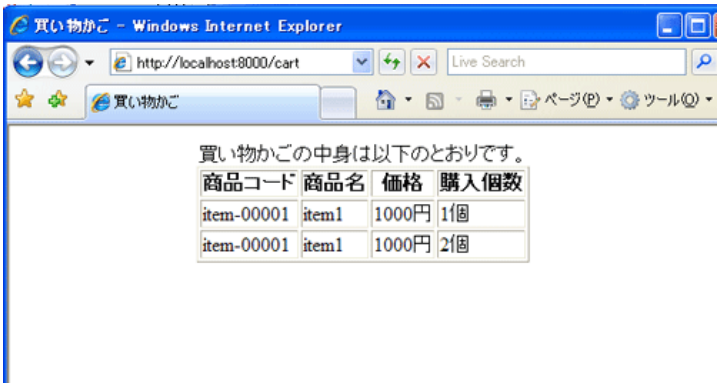
［リスト19］urls.py（買い物かご制御用ビュー関数とURLのマッピング）

```
urlpatterns = patterns('',
    〜省略
    (r'^cart', 'ecsite.itempage.views.do_cart'),
)
```

### 買い物かご機能の実行

それでは商品ページ「http://localhost:8000/item/1」にアクセスし「買い物かごに入れる」ボタンを押下してください。買い物かごの情報が表示されます。ブラウザの戻るボタンで戻り、購入個数を「2」を入力し「買い物かごに入れる」ボタンを再び押下してください。図13のように、前回「買い物かごに入れる」ボタンを押下した際の情報も一緒に表示されるはずです。これで前回分の情報がセッションに保存されていることが確認できました。

図13 買い物かご画面の表示




### セッション情報の保存ストレージについて

Djangoではセッション情報の保存先をいくつか用意しています。以下の表にセッション情報の保存先をまとめました。

セッション情報の保存先一覧

| セッション保存先 | 設定 | 説明 |
|---|---|---|
| データベース | SESSION_ENGINE= 'django.contrib.sessions.backends.db' | DBにセッション情報を保存する。Djangoのデフォルト設定 |
| ファイル | SESSION_ENGINE= 'django.contrib.sessions.backends.file' | ローカルファイルにセッション情報を保存する。 |
| キャッシュ | SESSION_ENGINE= 'django.contrib.sessions.backends.cache' | キャッシュシステム（後述）にセッション情報を保存する。 |

## キャッシュシステム

Djangoはアクセスの多いニュースサイトの開発をもとに生まれたフレームワークですので、サイトのパフォーマンスを向上させるための有効な仕掛けであるキャッシュ機能はとても充実しています。ここでは商品ページの表示機能にキャッシュの仕組みを導入します。

### キャッシュの単位

Djangoでは、キャッシュする情報の粒度をその使用用途に合わせて設定できるようになっています。以下の表にキャッシュの粒度とその説明をまとめました。本稿ではこの中からビュー単位でキャッシュする方法を紹介します。

キャッシュ方法の種類

| キャッシュの粒度 | 説明 |
|---|---|
| サイト単位 | サイト全体をキャッシュします。DBから情報を取得するが、コンテンツ自体の変更はあまりないサイトに向いているでしょう。 |
| ビュー単位 | URLごとに情報をキャッシュします。 |
| オブジェクト単位 | ビューの中で扱うオブジェクトをキャッシュします。 |
| テンプレートの断片単位 | テンプレートの中の一部分をキャッシュします。 |

**キャッシュ保存先と設定方法**

Djangoでは、キャッシュ情報の保存先を複数用意しているので適切なものを選択して使用することができます。以下の表に使用できるキャッシュの保存先を示します。キャッシュの保存先指定は、プロジェクトフォルダのsettings.py内のCACHE_BACKEND変数に「CACHE_BACKEND='memcached://172.19.26.240:11211'」のように指定します。このようにDjangoでは設定の変更だけで簡単にキャッシュの保存先を変更することができます。

キャプション

| 保存先 | 設定例 | 説明 |
|---|---|---|
| memcached | memcached://172.19.26.240:11211;172.19.26.242:11211/' | memcachedを使用。複数指定する場合、セミコロンで区切って指定する。パフォーマンス面で最も推奨される方式である。 |
| DB | db://my_cache_table | DBを使用。先にテーブルを作成する必要がある。「my_cache_table」に作成したテーブル名を指定する。 |
| File System | file://c:/codezine/cache（file;//キャッシュ保存先フォルダパス） | ローカルのファイルシステムを利用します。指定したフォルダの配下にキャッシュファイルが作成されます。 |
| Local-memory | 'locmem:/// | プロセスごとのメモリ内でキャッシュします。プロセス間でキャッシュを共有できないので実用的ではありません。 |
| Dummy | CACHE_BACKEND = 'dummy:///' | 開発時用の設定で、実際にはキャッシュしません。 |

**ビュー単位のキャッシュ**

それでは早速、ビュー単位のキャッシュを使用してみましょう。商品ページは商品属性が変わらない限り、表示内容はあまり変化がありませんので、キャッシュを使用しDBにアクセスする頻度を下げることで、システム全体の負荷を下げることができます。まずは、settings.pyファイルにキャッシュの保存先の指定をします。リスト20に設定を示しました。この設定では「c:/codezine/cache」というフォルダが作成され、その下にキャッシュファイルが作成されます。

［リスト20］settings.py

```
CACHE_BACKEND = 'file://c:/codezine/cache'
```

次に、キャッシュ対象のビュー関数item_page_displayにデコレータ記述を追加します。リスト21にソースを示します。デコレータに指定する単位は秒です。なお、キャッシュはURL単位でされますので、商品ごとにキャッシュがされます。例えば「http://hostname:8000/item/1」と「http://hostname:8000/item/2」では別々にキャッシュされます。

［リスト21］ビュー関数へのデコレータ定義

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15) # 単位は秒。15分キャッシュする。
def item_page_display(request,item_id):
    # ビュー関数の処理
```

## まとめ

Django チュートリアルとして前編・後編にまたがり、ECサイトを題材に商品ページの表示、商品データ登録用のバックエンド画面（管理インターフェイス）、商品検索フォーム、買い物かご機能、商品ページのキャッシュ機能と実装してきましたが、いかがだったでしょうか。これらの機能を簡潔な記述で、次々と実装できるのを実感されたことと思われます。

また、DjangoはWebアプリケーションフレームワークで必要な機能をフルスタックで用意していますので、使うまでの手順も簡単に済むため取っ付きやすいというのも魅力の一つです。エンジニアはDjangoを使うとすぐにプログラミングにとりかかることができ、すぐにアイデアを動くものとして実現していくことができるので、ものをつくるモチベーションがさらに高まり、開発生産性もスパイラルに上がっていきます。本稿がこれからPythonやDjangoに取り組む人にとっての一助となれば幸いです。