■本書内に記載されている会社名、商品名、製品名などは一般的に各社の登録商標または商標です。本書では®、™は明記しておりません。
■本書の出版にあたっては正確な記述に努めましたが、本書に基づく運用結果について、著者およびソフトバンククリエイティブ株式会社は一切の責任を負いかねますのでご了承ください。

©2007
本書の内容は著作権法の保護を受けています。著作権者・出版権者の文書による許諾を得ずに本書の一部、または全部を無断で複写・複製・転載をすることは禁じられております。

はじめに

この本は、オブジェクト指向のスクリプト言語Pythonの解説書です。Webアプリケーションの作成を通じて、Pythonの使い方や活用方法を学んでいただくのが本書のねらいです。立場的には、拙著**みんなのPython**の続編になります。インストール方法やPythonの文法など、基本的なことについても触れていますので、Pythonを知らない人でも楽しんでいただける内容になっています。

「みんなのPython」と同じく、本書の構成は**積み重ね式**になっています。学習を進めてゆくに当たって必要となる知識や技術を、基本的なことから順を追って解説しています。Pythonの基本はもちろん、Webの仕組みやWebアプリケーションの動作する原理からはじまり、比較的高度なWeb開発の手法にいたるまでを解説しています。本書が想定読者としているのは、Pythonに興味がある方と、Web開発に興味のある方の両方です。Web開発に興味のある方が、Pythonを使ってWebの仕組みを学べるような構成を心がけました。

Pythonはとてもシンプルで覚えやすいプログラミング言語です。プログラムを組むために覚えるべきことが少なく、思ったことをすぐに形にすることができるため、プログラミング初心者が使うのに最適な言語だと言えます。

プログラミングを学び始めて、自分の考えたことをコードとして書けるようになってくると、多くの方が決まった場所でつまずくのを目にします。関数、モジュール、クラスの作り方が分からない、という声をよく耳にします。本書では、そのような設計のコツを、Webアプリケーションの開発を通じてつかんでいただけるよう、実動するサンプルコードを多く掲載しました。

本書を通じ、Pythonを愛し、インデントを慈しむ方が増えることを願っています。

2007年10月某日
柴田 淳

CONTENTS

Part 1 PythonとWebアプリケーション

CHAPTER 01 Webアプリケーション概論 …… 2
01-01 アプリケーションとはなにか …… 2
入力と出力 …… 3
人間にとって使いやすいのがアプリケーション …… 4
アプリケーションとユーザインターフェース(UI) …… 5
・GUIアプリケーションとWebアプリケーション …… 5
・Webアプリケーションの台頭とスクリプト言語 …… 6

01-02 Webの基本的な仕組みを知る …… 6
HTMLの概要 …… 7
・HTMLの文法 …… 7
・タグは命令の一種 …… 7
・タグの構造 …… 8
Webでユーザインターフェースを作る …… 9
・WebアプリケーションのUI—フォーム …… 9

01-03 HTTPの概要 …… 10
クライアントとサーバ …… 11
リクエストとレスポンス …… 12
・リクエスト—サーバへの要求 …… 13
・レスポンス—サーバからの応答 …… 14
静的出力と動的出力 …… 14
・プログラムでレスポンスを作って返す動的出力 …… 15
ユーザとWebアプリケーションのやりとり …… 15

CHAPTER 02 復習!! Python …… 18
02-01 PythonとWebアプリケーション …… 18
Pythonのインストール …… 19
・WindowsにPythonをインストールする …… 20
・MacOS XにPythonをインストールする …… 21
・LinuxにPythonをインストールする …… 21

02-02 Pythonの基礎 …… 22
・Pythonの起動 …… 23
組み込み型 …… 25
・数値型 …… 25
・文字列型 …… 25
・ユニコード文字列型 …… 26
・リスト …… 26
・辞書(ディクショナリ) …… 27
・タプル型 …… 27
・その他の組み込み型 …… 28
フロー制御 …… 28
・条件分岐(if文) …… 29
・シーケンスを使った繰り返し(for文) …… 30
・条件を使った繰り返し(while文) …… 30
関数定義と呼び出し …… 30
・引数の定義 …… 31
・関数の戻り値とアンパック代入 …… 31
・関数と名前空間 …… 32
・関数の呼び出し …… 32
モジュールの活用 …… 32
・モジュールのインポート …… 33
・モジュールの作成 …… 33

02-03 Pythonのオブジェクト指向機能 …… 34
・すべてがオブジェクト …… 35
クラスを定義する …… 35
・クラスインスタンスの生成 …… 36
・第1引数self …… 36
・特殊メソッド …… 37

02-04 ユニコードと日本語処理 …… 37
・ユニコード型と文字列の境界 …… 37
・他のエンコードからユニコード文字列への変換 …… 38
・ユニコード文字列から他エンコードへの変換 …… 39
・プログラム内部での日本語の扱い …… 39

CHAPTER 03 PythonでWebサーバを作る …… 40
03-01 SimpleHTTPServerを使う …… 41
HTMLファイルを扱う …… 42
・Webサーバを起動するスクリプト …… 43

03-02 CGIHTTPServerを使う 44
CGIHTTPServerを起動するスクリプト 44
Webサーバにプログラムを設置する 45
簡単なプログラムを動かす 46

CHAPTER 04 Webアプリケーションに値を渡す 49
04-01 URLを使ってWebサーバに命令を渡す 49
プログラムで値を受け取る 50
クエリのキーをスマートに扱う 51
・「13日の金曜日」を探すWebアプリケーション 52

04-02 フォームの処理 55
フォームを使ってWebサーバに値を渡す 55
フォームの動的出力 58
2つのmethod：GETとPOST 60
・GETメソッドとPOSTメソッドの違い 61
いろいろなフォーム・コントロール 61
・テキストフィールド 62
・サブミットボタン 62
・リセットボタン 63
・ラジオボタン 63
・チェックボックス 64
・メニュー 64
・テキストエリア 65

04-03 データ型の変換 66
・文字列型から整数型への変換を行う 67
・文字列型から浮動小数点型への変換を行う 67
・数値型に変換できるかどうかを調べる 67
・ユニコード型への変換 68
縦書き変換プログラム 69

04-04 クエリとリスト 72

CHAPTER 05 HTTPの詳細 76
05-01 レスポンスとして送られる文字列 77
ステータスコード 77
レスポンスのヘッダ 78

05-02 リクエストとして送られる文字列 79
リクエストラインとヘッダ 79
HTTPメソッド 81
・GETメソッド 81
・クエリとURLエンコード 82
・POSTメソッド 83
・GETメソッド、POSTメソッドの利点、欠点 84

CHAPTER 06 RSSリーダーを作る 86
06-01 リクエストの処理 86
Requestクラスを実装する 87
Requestクラスの使い方 88

06-02 レスポンスの処理 89
Responseクラスを実装する 89
・ヘッダの追加用メソッド 91
・そのほかの処理 93

06-03 簡易RSSリーダーを作る 94
RSSを読み込む 94
Webアプリケーションを作る 95
Webアプリケーション開発とヒアドキュメント 98

CHAPTER 07 Webアプリケーションとデータの保存 100
07-01 Webアプリケーションとセッション 100
標準ライブラリを使ってデータを保存する- pickleを使う 102
・データの保存とシリアライズ 102
・標準モジュールpickleを使ってシリアライズを行う 103
Webアプリケーションでデータを保存する 104
pickle利用とマルチスレッド 107

07-02 データベース概論 109
リレーショナルデータベースとは 109
テーブルとカラム 110
カラムとデータ型 112

・IDとシリアル型 112
SQL：データベース用の問い合わせ言語 113
・INSERT文（データの登録） 113
・SELECT文（データの選択） 114
・UPDATE文（データの更新） 114
・DELETE文（データの削除） 115

07-03 Pythonとデータベースの連携 115
SQLiteをPythonから利用する 116
・PythonとSQLiteを接続する 116
DBAPIを使う 117
・コネクションオブジェクトを作る 118
・カーソルを使う 119
SQLの動的生成 120
・プレースホルダを使う 121
Webアプリケーションでデータを保存する - データベース編 122

Part 2 効率的なWebアプリケーション開発 127

CHAPTER 08 効率的なWebアプリケーション開発とは 128
08-01 アプリケーションと開発効率 129
素朴な手法の問題点 129
モジュール化、クラス化による開発の効率化 130
役割分担による開発の効率化 131

CHAPTER 09 Pythonとテンプレートエンジン 133
09-01 テンプレートエンジンとは 133
09-02 テンプレートエンジンの動く仕組み 134
09-03 標準モジュールを使ったテンプレートエンジン 136
Templateクラスの使い方 137
Templateクラスを活用してブックマーク管理Webアプリを作る 138
・テンプレートファイルを作る 139
・Webアプリケーションのロジックを作る 140
・ブックマークリストを埋め込む 143
テンプレートエンジンに求められる機能と分類 144
・ループ 145
・条件分岐 145
・テンプレートの部品化 146
・マスターテンプレートとスロット 147
・コードブロックの埋め込み 148
・テンプレートエンジンの分類 149

09-04 Pythonでテンプレートエンジンを作る 150
テンプレートエンジンの仕様を決める 150
実装の指針 151
・初期化部分の実装 152
・レンダリング処理を実装する 153
・パターンの置換を処理する 154
・条件分岐の処理 157
・ループの処理 158
・そのほかの処理 159
ブックマーク管理Webアプリを書き換える 163

CHAPTER 10 O/Rマッパーを使ったデータベースの操作 167
10-01 テーブルとクラスの関係 167
テーブルとクラスの違い 168
・データを登録する場合 169
・特定のデータを更新する場合 169
O/Rマッパーとは 171

10-02 シンプルなO/Rマッパーを作る 172
設計の指針 172
テーブルを作成する 174
テーブルにデータを追加（INSERT）する 176
インスタンスオブジェクトの初期化 178
データを更新する機能（UPDATE） 180
テーブルから条件に合うデータを取り出す機能（SELECT） 181
そのほかの処理 184

10-03 O/Rマッパーの利用例 185

CHAPTER 11 RSSリーダーを作る その2 187
11-01 RSSリーダーの機能追加 187

O/Rマッパーのクラスを作る 189
・動作チェック 190
巡回用RSSの一覧ページを作る 190
・テンプレートファイルの作成 191
編集用フォームを作る 193
新規登録用フォームを作る 197
RSSの一覧ページを作る 200

CHAPTER 12 バリデータとウィジェット―フォーム処理の抽象化 203
12-01 バリデータの利用 203
メールアドレスのバリデーションチェック 204
バリデータとは 205

12-02 バリデータを作る 205
バリデータの戻り値と例外 206
バリデータの抽象クラスと初期化が不要なバリデータ 207
入力値が整数で、指定された範囲にあることを調べるバリデータ 209
正規表現を使ったバリデータ：メールアドレス・URLのチェック 209

12-03 ウィジェットの利用 211
WebアプリケーションとCRUDフォーム 212
フォーム処理の抽象化 214
ウィジェットとは 215

12-04 ウィジェットを作る 216
ウィジェットの抽象クラスを作る 217
テキスト入力フォーム用のウィジェットを作る 220
メニュー・ラジオボタン用のウィジェットを作る 221
サブミットボタンとリセットボタン 223
フォームをまとめるウィジェットを作る 224
ウィジェットとバリデータを使ったサンプルプログラム 226
・テンプレートの作成 229
・Webアプリケーションの作成 229

CHAPTER 13 Webアプリケーションサーバを使った開発 232
13-01 URLと遷移 232
CGIとURL 233

13-02 Webサーバの動く仕組みを理解する 234
ファイルをレスポンスとして送信する 234
Webサーバ内でプログラムを起動する 236
Webサーバの基本的な働き 237

13-03 Webアプリケーションサーバを作る 238
Webアプリケーションサーバの仕様を決める 239
クラス定義とリクエストを受け取るメソッド 240
・do_POST()メソッド 241
リクエストに合わせて関数を呼び出す 242
サーバの起動 245
WebアプリケーションサーバとCGIの違い 246
・countertest.pyの起動 247

CHAPTER 14 Webアプリケーションと認証 249
14-01 認証の基本 249
Webアプリケーションでの認証 250
認証状態の継続 250

14-02 BASIC認証 251
BASIC認証の仕組み 251
BASIC認証の継続 252
認証情報の暗号化方式 253

14-03 ダイジェスト認証 254
メッセージダイジェストとハッシュ 254
ダイジェスト認証の仕組み 255
ダイジェスト認証の継続 257

14-04 フォームを使った認証 257
フォーム認証の遷移 257
フォーム認証での認証状態の継続 258
PythonとCookie 260

14-05 フォーム認証の機能を作る 261
認証フォーム一式を作る 262
ログインチェック用の仕組みを作る 265
フォーム認証を試す 268

CONTENTS

CHAPTER 15 **Webアプリケーションとセキュリティ** …… 270

15-01 **Webアプリケーションのセキュリティホール** …… 270

セキュリティホールの例 …… 271

15-02 **セキュリティホールへの対処** …… 273

クロスサイトスクリプティング(XSS) …… 274

クロスサイトスクリプティングの対策 …… 275

クロスサイトスクリプティングとテンプレートエンジン …… 275

SQLインジェクション …… 276

・SQLインジェクションの対策 …… 277

クエリに関連するセキュリティホール …… 277

CHAPTER 16 **RSSリーダーを作る その3** …… 279

16-01 **拡張版RSSリーダーの仕様を決める** …… 279

既存モジュールの変更 …… 280

16-02 **RSSリーダーの作成** …… 281

追加フォームを作る …… 283

編集フォームを作る …… 285

編集用URL一覧ページを作る …… 286

RSS一覧ページを作る …… 287

テンプレートの作成 …… 288

RSSリーダーの実行 …… 289

16-03 **機能拡張のためのヒント** …… 290

ユーザ登録機能 …… 290

RSSの内容をデータベースに登録する …… 291

APPENDIX **Webアプリケーションフレームワークの活用** …… 294

A-01 **Webアプリケーションフレームワークとは** …… 294

A-02 **Django** …… 296

A-03 **TurboGears** …… 300

A-04 **Plone** …… 304

A-05 **WSGI** …… 307

Part 1

Pythonと
Webアプリケーション

CHAPTER 01

Webアプリケーション概論

コンピュータやプログラミングの世界では、よく**ドッグイヤー**という言葉を使います。犬は1年で人間の7年分年を取ります。ふつう7年かかるような変化が1年で起こるような「変化や技術革新の速さ」の比喩としてこの言葉が使われます。そのくらい変化の激しい世界に、私たちは身を置いているわけです。

コンピュータの世界では、次から次へと新しい言葉が現れます。**Webアプリケーション**という言葉も、比較的新しく使われるようになった言葉の1つです。WebというのはWebブラウザやWebサイトのWebです。アプリケーションとは、コンピュータ上で動くソフトウエアを指す言葉です。

簡単に言うと、**Webブラウザを使って動くアプリケーション**をWebアプリケーションと呼んでいます。Webアプリケーションは、Webブラウザがあれば場所を選ばず動き、開発が手軽であることから、最近とても注目されています。Pythonのような軽量言語（LL）はWebアプリケーションの開発に向いていると言われています。多くのWebアプリケーションがPythonを使って作られています。

Webアプリケーションとはなにかについて解説する前に、アプリケーションについて簡単に解説しましょう。

01-01　アプリケーションとはなにか

コンピュータはソフトウエアを使って動かします。ソフトウエアという言葉が指す範囲はとても広く、たとえばPythonで書いたプログラムもソフトウエアの1つですし、WindowsやLinuxのようなオペレーティングシステム（OS）もソフトウエアの一種です。ソフトウエアの中でも、ワープロや表計算ソフトのように、特別な目的を果たすために作られたソフトウエアを**アプリケーション**と呼びます。

アプリケーションにはいろいろな種類があります。ワープロや表計算ソフトのように高機能なアプリケーションもありますし、付箋アプリケーションや計算機のようにシンプルなものもあります。拙著「みんなのPython」の最後では、Pythonを使って簡単なタイマーを作りました。35行というとても短いPythonプログラムですが、これも立派なアプリケーションの一種です。

図01 「みんなのPython」で作成したタイマー

入力と出力

高機能なものからシンプルなものまで、いろいろな種類のあるアプリケーションですが、ほぼすべてのアプリケーションには以下のような共通した特徴があります。

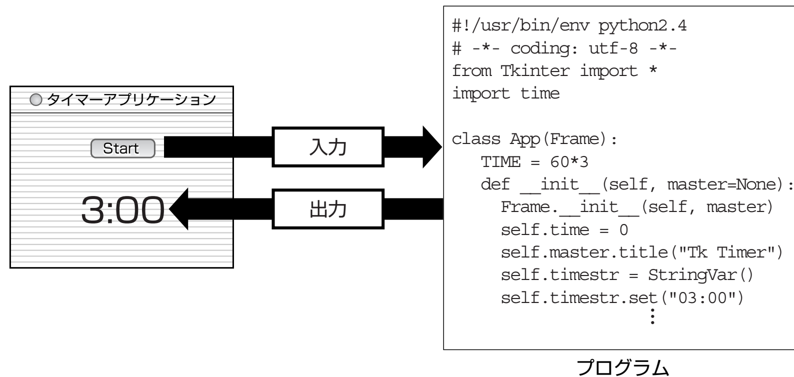
A：人間が指示を与えるための仕組みがある（入力）
B：指示に従ってソフトウエアが処理し、結果を返す（出力）

たとえば、Pythonで書いたタイマーを例に取りましょう。タイマーの基本的な機能は、次のようになります。

1：人間がマウスを使ってボタンを押す
2：タイマーが動き始め、残り時間が表示される

この場合は、1が**入力**（A）に、2が**出力**（B）になります。こう見ると、入力に対してプログラムが反応し、結果を返すということがアプリケーションの基本的な動作である、ということが分かります。

図02 アプリケーションの入力と出力



人間にとって使いやすいのがアプリケーション

入力に対して出力を返す、というのはすべてのソフトウエアが持つ共通した特徴です。このようにしてみると、アプリケーションとプログラムは似た性質を持っていると言えます。Pythonの関数やメソッドであれば、入力は**引数**に、出力は**戻り値**になります。

ただし、プログラムで扱う入力や出力は、数値や文字列のような「生のデータ」であることがほとんどです。そのような生のデータは、人間にとって扱いやすいとは言えません。たとえば、Eメールを送信するときにいちいちPythonの関数を使わなければならないとしたら、よほどPythonが好きな人でない限りウンザリすることと思います。もしそのような方法でしかEメールが使えないとしたら、Eメールは今ほど多くの人に利用されていないでしょう。

Eメールアプリケーションを使えば、もっと手軽にメールを送信できます。

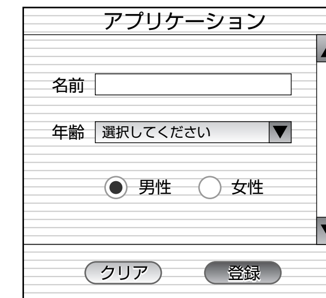
- アドレス帳からメールアドレスを指定してメールを作る
- ウインドウに文章を打ち込む
- ボタンを押してメールを送信する

このように、多くのアプリケーションは誰でも使えるような操作方法を備えています。同様に、操作や処理の結果についても分かりやすく工夫されています。アプリケーションは、人間にとって分かりやすいように入力と出力を工夫したソフトウエアと言えます。

アプリケーションとユーザインターフェース（UI）

アプリケーションが共通して持つ特徴として**人間が指示を与えるための仕組み**があると書きました。この仕組みのことを**ユーザインターフェース**（User Interfase）と呼びます。または英単語の頭文字を取ってUI（ユーアイ）とも呼ばれます。アプリケーションに表示されるボタンやメニューなどはUIの一種です。ワープロのウインドウには、文字を打ち込む領域、文字の大きさや割り付けを変えるボタンなど、多くのUIが配置されています。ワープロのUIはとても複雑なUIの例と言えます。

図03 ユーザインターフェースの例



GUIアプリケーションとWebアプリケーション

UIや、UIによる操作結果をどのように表示するかによってアプリケーションを分類することがあります。コンピュータの画面（グラフィックス画面）を直接操作して表示するUIを**グラフィカル・ユーザインターフェース**（GUI）と呼びます。GUIを使ったアプリケーションをGUIアプリケーションと呼びます。ワープロや表計算ソフトなど、皆さんにもなじみのあるアプリケーションの多くはGUIアプリケーションです。

一方、Webブラウザを使ってUIや操作の結果を表示するアプリケーションを**Webアプリケーション**と呼びます。会員制Webサイトの登録フォームなど

もWebアプリケーションの一種です。ブログもWebアプリケーションです。Webを使って記事を投稿したり、管理するために、Webアプリケーションとしての機能を持っています。

GUIを使ったアプリケーションは、一般的にOSに密接なつながりを持っています。特にグラフィックス機能を活用してUIを表示するための手法は、OSによってまちまちです。このため、たとえばWindows用に作ったGUIアプリケーションをMacやLinuxで動かすことはできません。多くの場合、アプリケーションを複数のOSで動かそうとすると、それぞれのOSで動く別のアプリケーションを作らなければならないのです。

Webアプリケーションの特徴は**どこでも動く**ということです。Webブラウザとインターネットさえあれば、WindowsやMac、Linux、または携帯電話などいたるところで利用できます。Webには、HTTPのような通信方法（プロトコル）やHTMLといった標準的な仕様があります。標準を守りさえすれば、誰でも手軽に、あらゆるところで動くWebアプリを作ることができます。

Webアプリケーションの台頭とスクリプト言語

Webアプリケーションで実現できる機能は、GUIアプリケーションに比べて制限があると言われてきました。しかし近年、Ajaxのような技術によってWebアプリケーションの自由度はとても高まっています。どこでも動き、高機能なアプリケーションを作ることができるWebアプリケーションに、多くの開発者が注目しています。

Webアプリケーションでは、たくさんの文字列操作を行います。そのため、スクリプト言語がWebアプリケーションの開発に利用されてきました。もちろん、PythonもWebアプリケーションを作るのに向いたプログラミング言語の1つです。世界中でPythonを使って作られたWebアプリケーションが活躍しています。

01-02　Webの基本的な仕組みを知る

本書では、Pythonを使ってWebアプリケーションを開発するための手法について解説します。具体的な手法について解説する前に、Webアプリケーションを作る上で知っておくべき基本的なことがらについて解説します。

HTMLの概要

本書を読んでいる皆さんは、HTMLについてある程度知っているか、少なくとも聞いたことがあると思います。HTMLは、Webブラウザなどで表示するWebページを作るために利用します。Webアプリケーションでは、処理の結果などを表示するためにHTMLを活用します。

Webページに表示する文字の大きさや配置、画像の種類や位置などを決めるための規則があり、規則を定めているのがHTMLです。ここでいう規則は**文法**と言い換えてもよいでしょう。後半の「ML」は**マークアップ言語**（Markup Language）の略です。HTMLはPythonと同じ言語の一種なのです。HTMLには文法があり、Pythonのインデントブロックのような構造があります。

HTMLの文法

タグを使って、文字や文書に目印を付ける、ということがHTMLの基本的な文法です。**<と>**で文字列を囲んだものがタグです。タグには「開始タグ」と「終了タグ」の2種類があります。たとえば、HTMLの中に以下のように書かれていたとします。

```
<title>Python</title>
```

この部分は、「Python」という文字列を文書の中で特別な性質を持つように目印を付けています。「<title>Python</title>」という部分は、囲まれた文字列を文書のタイトルとして扱うようにする、一種の命令と見ることができます。<title>というタグの他にも、いろいろな意味を持つタグが用意されています。

タグは命令の一種

もう少し複雑なHTMLの例を見てみましょう。

List01 htmlSample.html

```
<html>
<body>
    <h1>Pythonのリンク</h1>
    <a href="http://www.python.org/">python.org</a>
</body>
</html>
```

図04 Webブラウザでの表示

<h1>タグは、囲まれた文字列を「レベル1の見出し」として目印を付ける働きがあります。見出しは目立った方がよいので、囲まれた文字列は大きく表示されます。

<a>タグの中を見てください。タグの中には別の文字列が見えます。これは**アトリビュート**と呼ばれる部分です。hrefはアトリビュート名と呼ばれています。<a>タグはハイパーリンクを埋め込むために利用します。アトリビュートにはリンク先のURLを指定します。

<a>タグはリンクを埋め込むという命令、ととらえることができます。アトリビュートを使うと、この命令に意味を追加することができます。タグが命令だとすると、アトリビュートは引数のようなものだと言えます。

タグの構造

このHTMLでは、タグが**入れ子**になっています。<html>タグが全体を囲っていて、その下にさらに<body>タグがあります。ある要素が他の要素を囲むことによって、タグ同士が**親子関係**を持ちます。<h1>タグ、<a>タグは<body>タグの子供に当たります。

このように、タグで目印を付けて文書に意味を持たせるのがHTMLの書き方の基本です。タグをどのように書くべきかを定義した文法のことをHTMLと呼ぶのです。

Webでユーザインターフェースを作る

HTMLには基本的なUIを作るための命令（タグ）が用意されています。メニューやボタンなど、一般的なアプリケーションでよく見かけるUIはほとんど揃っています。アプリケーションにとってUIは重要な要素です。Webアプリケーションを作る場合は、HTMLを使ってUIを作ります。

WebアプリケーションのUI―フォーム

HTMLには**フォーム**（form）と呼ばれる仕組みが備わっています。Webアプリケーションで利用するUIはフォームを活用して作ります。

簡単なフォームの例を見てみましょう。

List02 formSample.html

```
<html>
<body>
<form>
    ニックネーム ：
    <input type="text" name="nickname" />
    好きな言語 ：
    <select name="faborite_language">
        <option>Python</option>
        <option>Ruby</option>
        <option>Perl</option>
        <option>PHP</option>
    </select>
    <input type="submit" value="送信" />
    <input type="reset" value="取り消し" />
</form>
</body>
</html>
```

図05 Webブラウザでの表示

このHTMLの中には、<input>タグや<select>タグが見えます。このタグが、Webページ上にUIの部品を表示するための命令になっています。<input>タグには、typeというアトリビュートが見えます。このアトリビュートに指定する文字列によって、違った種類のUIが表示されています。また、ボタンのタイトルはvalueアトリビュートとして指定していることに注目してください。このように、タグごとに利用するアトリビュートが決まっているのです。

<select>タグはメニューを表示するための命令です。このタグは、子供として<option>タグを持っています。<option>タグで囲まれた文字列はメニューの項目となります。

UIの部品を表示しているタグには、nameというアトリビュートが付いています。このアトリビュートは、UIの部品を特定するために利用する重要なアトリビュートです。

<form>タグはUIの部品全体を囲む形で記述します。UI部品の親となるのが<form>タグです。

Webアプリケーションで実際に利用する<form>タグには、アトリビュートを何種類か記述します。アトリビュートの種類や意味についてはのちほど解説します。ここでは、HTMLを使ってUIを表示するための方法について、概要を理解できればOKです。

HTMLを使うことによってアプリケーションの2つの重要な要素を作ることができることを分かっていただけたでしょうか。フォームを使ってUIを作り、結果をHTMLとして表示することで、アプリケーションの基本的な機能を満たすことができるのです。

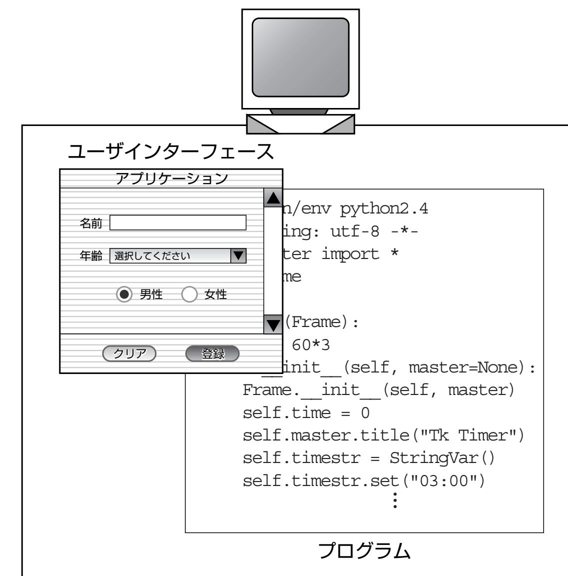
01-03 HTTPの概要

ここでは、Webの内部的な仕組みについて基本的なことがらを解説します。Webアプリケーションが動くとき、実際にどのような仕組みでデータのやりとりがされているのかについて、概要を解説します。

クライアントとサーバ

前の節で、UIをどのように表示するかによってアプリケーションを分類することがあると解説しました。表計算、Eメールアプリケーションのような GUIアプリケーションでは、ユーザの操作を受け取るUIと、操作に対応する処理を実行したり結果を表示するプログラムが一体化していることがほとんどです。UIを表示したり、処理を実行するプログラムは1つのパソコンの中で動きます。このようなアプリケーションを**スタンドアロン型アプリケーション**と呼びます。スタンドアロン（stand alone）とは「独立した」、「一人で立つ」というような意味の英語です。

図06 スタンドアロン型アプリケーション



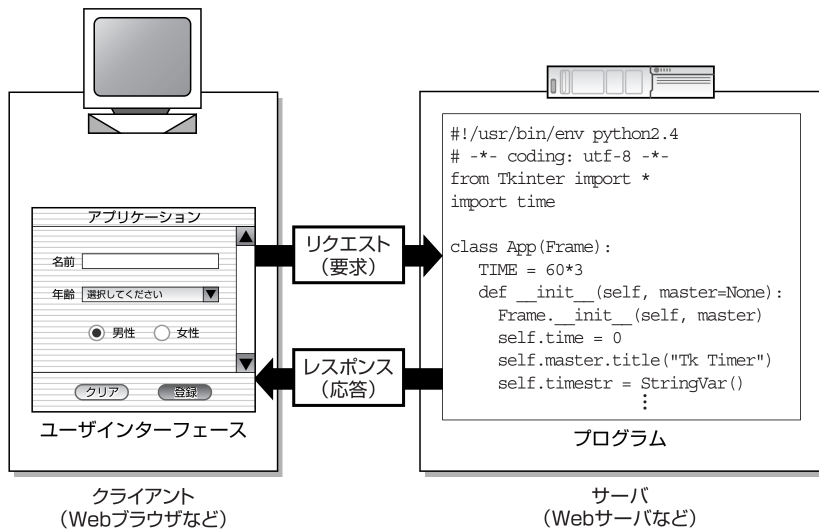
これに対して、WebアプリケーションではUIや結果を表示する部分と、データを受け取って処理を行う部分が分かれています。このようなアプリケーションを**クライアント・サーバ型アプリケーション**と呼びます。

ここで言うクライアントとは、特定の機能を専門に受け持つアプリケーシ

ョンを指します。Webアプリの場合は、HTMLで書かれたUIなどを表示するWebブラウザがクライアントに相当します。

サーバには、処理を実行するプログラムを置きます。ユーザが実際に操作をするのはクライアントになります。クライアントとサーバの間では通信が行われることになります。アプリケーションが動くときに通信をするかしないかが、スタンドアロン型のアプリケーションとWebアプリのようなクライアント・サーバ型アプリケーションの決定的な違いです。

図07 クライアント・サーバ型アプリケーション



サーバは、UIに入力された内容を通信で受け取ります。必要があれば、処理の結果を通信でクライアントに戻します。つまり、通信には2つの方向があるわけです。

リクエストとレスポンス

クライアントからサーバに向かう通信を**リクエスト**（要求）と呼びます。サーバからクライアントへの通信を**レスポンス**（応答）と呼びます。クライアントから処理を要求し、サーバが処理結果を応答として返す。これがWebアプリケーションの基本的な動作原理です。Webアプリケーションだけでなく、Webサイトもこのような仕組みを使ってWebページを表示しています。

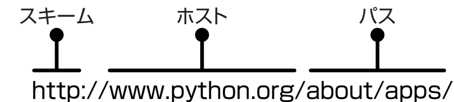
リクエスト―サーバへの要求

リクエスト（要求）とは、Webの通信のうちWebブラウザ（クライアント）からサーバに送られる通信のことを指します。こう書くと難しく感じるかもしれません。でも、ほとんどの方は、知らず知らずのうちにリクエストを使いこなしています。

Webページを表示するときに入力する**URL**はリクエストの一種です。URLの書き方には規則があります。規則に従ってURLを分割すると、サーバに対してどのようなリクエストが送られているかを知ることができます。

以下のようなURLを例に取って、URL文字列の意味について簡単に解説してみましょう。

図08 URLはいくつかの部分に分けることができる



URLは、大まかに分けて前半と後半に分かれます。前半は**スキーム**、**ホスト**と呼ばれる要素から構成されています。スキームはどのような手法を使ってリクエストを送るかを指定する部分です。ホストは、リクエストを送る先がインターネット上のどこにあるかを指定する部分です。Webページを表示するWebブラウザは、前半部分を見てリクエストを送る方法と場所を解釈します。

URLの後半部分は**パス**と呼ばれています。このURLでは、Pythonで作られているアプリケーションの一覧が書かれたHTMLを送るよう、パスを指定しています。

URLのパスを解釈し、HTMLを応答として返しているのは**Webサーバ**と呼ばれる一種のプログラムです。つまり、URLのパスを使ってWebサーバに命令を伝えているわけです。パスを変えると、Webサーバは違う種類のHTMLを送り返します。与えられたパスを処理できない場合には、Webサーバはエ

ラーを返します。

レスポンス―サーバからの応答

クライアントがリクエスト（要求）を送ると、サーバがリクエストを解釈します。リクエストを解釈した結果を、レスポンス（応答）として返します。Webサーバがリクエストを受け取った場合は、パスをファイルの**階層**と解釈します。その結果、パスに相当する場所に存在するHTMLファイルなどをレスポンス（応答）として返します。レスポンスを受け取ったクライアントは、HTMLを表示します。

Webページはこのような流れで表示されています。この流れはWebアプリケーションを作るときでも同様です。

静的出力と動的出力

リクエストに反応するWebサーバは一種のプログラムです。多くの場合は、事前にHTMLファイルなどをサーバ上に設置しておきます。リクエストに合わせて事前に設置したファイルを探し、レスポンスとして返します。一般的なWebサイトであればこれで十分なのですが、Webアプリケーションのようなものを作る場合には困ったことが起こります。

たとえば、西暦と月の情報を与えてカレンダーを表示するWebアプリケーションを作るとします。Webブラウザ上にカレンダーを表示するわけです。URLを使うと、Webサーバに対して命令を与えることができます。たとえば、2007年12月のカレンダーを表示したい場合には、以下のようなURLを与えればよさそうです。

http://<ホスト>/2007/12.html

サーバ上に、西暦ごとにディレクトリを、月ごとにカレンダーを表示するHTMLファイルを用意するわけです。このようにすれば、URLの中のパスを活用して、Webサーバに命令を与えて必要なレスポンスを得ることができます。

このように、事前に作っておいたファイルを使ったレスポンスを**静的出力**と呼びます。ただし、この方法だとディレクトリやファイルをたくさん用意しなければなりませんし、そもそもアプリケーションとは呼べないかもしれません。

プログラムでレスポンスを作って返す動的出力

Web上でこのようなことを実現する場合、大抵はWebサーバ側で動くプログラムを利用します。プログラムでカレンダーを表示するためのHTML文字列を作り、レスポンスとして返すわけです。Webブラウザでレスポンスを表示するためには、カレンダーはHTMLになっている必要があります。プログラム側では、文字列処理を繰り返してカレンダーに相当するHTMLを組み立てて、応答として返すことになります。このように、プログラムを使って組み立てたレスポンスを**動的出力**と呼びます。

動的出力を実現するために利用されるのが**CGI**と呼ばれる仕組みです。また最近では、より高度な**アプリケーションサーバ**と呼ばれる仕組みが使われることが多くなってきました。もちろん、PythonでCGIを作れますし、Pythonを活用できるアプリケションサーバは何種類もあります。

Webアプリケーションでは、UIからの指示に従ってさまざまなレスポンスを返す必要があります。Webアプリケーションでは、いろいろな場面で動的出力を活用します。

ユーザとWebアプリケーションのやりとり

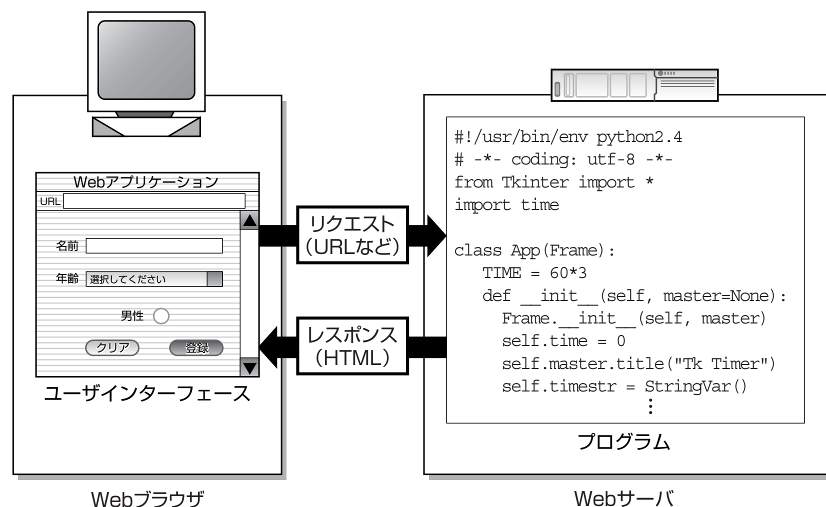
さて、ここまでWebアプリケーションを作るために知っておくべき基本的な技術について解説してきました。ここでは、それぞれの要素を見返しながら、Webアプリケーションが実際にどのように動いているのかを簡単に見ていきたいと思います。

まず、アプリケーションには命令を与えるためのUIが必要です。Webアプリケーションでは HTMLを使ってUIを作ります。HTMLで作ったUIをWebブラウザで表示するわけです。

UIからの情報はリクエストという形でサーバに送信されます。リクエストには、URLの他にもいろいろな情報を載せることができます。送られるのは主に文字ベースの情報です。

サーバ上にはプログラムが設置されています。Webアプリケーションの場合、実際にリクエストを受け取るのはこのプログラムです。

図09 クライアントとWebアプリケーション



プログラムでは、受け取ったリクエストを元にレスポンスとなる文字列を組み立てます。レスポンスはWebブラウザが受け取り、操作の結果としてWebページを表示します。

ここで、いったんWebアプリケーションの特徴をまとめてみましょう。

●表示にHTMLを利用する

UIや結果の表示にはHTMLという一種の言語を利用します。

●クライアントとサーバが分かれている

UIを表示したり、プログラムにリクエストを与えるためのクライアント、実際に処理を行い、レスポンス作るサーバに分かれています。

●レスポンスを得るためにはリクエストが必要

リクエストとレスポンスはペアになっている必要があります。画面を書き換えるためにはレスポンスが必要です。Webアプリケーションでは、画面を書き換えるためにはリクエストを送る必要があります。

●やりとりするのはテキストデータ

Webアプリケーションで扱うのは主にテキストデータです。リクエストもほとんどはテキストです。レスポンスとなるHTMLもテキストデータです。

このように、クライアントとサーバの間で、**リクエストとレスポンスを繰り返しながら処理をしていく**のがWebアプリケーションの基本的な動作となります。ユーザとWebアプリケーションが、UIを通じて対話をしながら処理を進めていくわけです。

CHAPTER 02

復習!! Python

この節では、Pythonをインストールし、Webアプリケーションを作る方法について解説します。Pythonを使うと、とても手軽にWebアプリケーションを作り、実際に動作を試すことができます。

02-01 PythonとWebアプリケーション

Webアプリケーションでは、HTMLというテキストを使って結果やユーザインターフェース（UI）を表示します。また、ユーザが利用するクライアント（Webブラウザ）から、プログラムが動いているWebサーバに送られるデータの多くは文字列です。このため、Webアプリケーションの開発では**非常に多くの文字列処理を実行**します。

Pythonのようなスクリプト言語を使うと、文字列を多く処理するプログラムを手早く作ることができます。つまり、PythonはWebアプリケーションの開発に向いたプログラミング言語なのです。

また、PythonにはWebアプリケーションを作るときに便利に活用できる**標準モジュール**が搭載されています。標準モジュールは、Pythonの機能を拡張するために利用します。Webアプリケーションを作るとき、よく実行する処理の多くがすでにPythonに搭載されているわけです。そのようなモジュールを活用することで、Webアプリケーションをより簡単に作ることができます。また、標準モジュールにはPythonを使って簡単なWebサーバを作るための機能も搭載されています。このような機能を活用すれば、わざわざWebサーバをインストールしなくても、今皆さんが使っているパソコンでWebアプリケーションの機能を試したり、Webアプリケーションを作ることができるのです。

本書の前半では、Pythonだけを使ってWebアプリケーションを作るための方法を解説します。普段使っているパソコンにPythonをインストールするだけで、Webアプリケーションを試し、仕組みを学ぶことができるように配慮しています。

Webアプリケーションはクライアント・サーバ型のアプリケーションです。そのため、実行にはWebサーバが必要です。本書では、Pythonと標準ライブラリを使って簡単なWebサーバを作り、皆さんが普段使っているパソコンで動かしながらWebアプリケーションについて学んでいきます。

実際にWebアプリケーションを作る前に、Pythonのインストール方法について簡単に解説します。Pythonをインストールしていない方は、本書を読み進める前にぜひPythonをインストールしてください。

インストール方法を解説した後は、Pythonの基本的な機能や文法についても簡単に触れます。

Pythonのインストール

Pythonは無料で利用できるプログラミング言語です。Windows、Linux、MacOS XなどいろいろなOS環境でPythonを動かすことができます。

LinuxやMacOS Xでは、最初からPythonがインストールされています。Windowsでは、WebサイトからPythonをダウンロードし、インストールすることでPythonを使えるようになります。

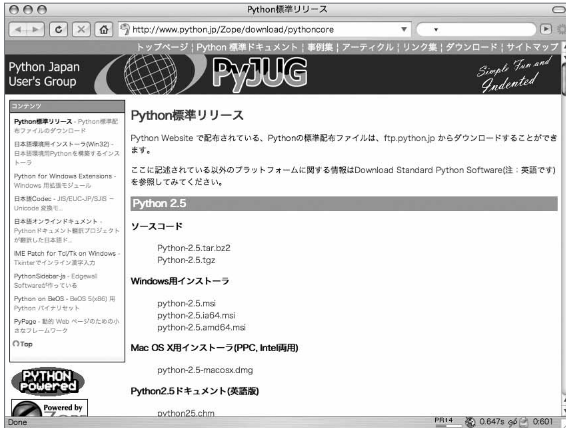
本書ではPythonを使ってWebアプリケーションを作るための手法について解説します。本書をこの先読み進める前に、ぜひPythonをインストールしてください。普段使っているパソコンにPythonをインストールするのが手軽です。

本書では、**Python 2.5以上のバージョン**を対象に解説を行っています。新しくPythonをインストールする場合は、最新のバージョンを使ってください。

日本Pythonユーザ会のWebサイトの中に「Python標準リリース」というページがあります。ここから、インストールする環境に合ったインストーラをダウンロードしてPythonをインストールしてください。

URL 日本Pythonユーザ会
http://www.python.jp/Zope/download/pythoncore

図01 日本Pythonユーザ会のサイト



WindowsにPythonをインストールする

「Python標準リリース」というWebページに「Windows用インストーラ」というリンクがあります。このリンクをクリックして、Windows用のインストーラをダウンロードしてください。ダウンロードしたら、インストーラを起動します。何回か「Next」ボタンをクリックすると、Pythonのインストールが始まります。

図02 Pythonインストーラ

WindowsにPythonをインストールすると、スタートメニューにPython用の項目が追加されます。スタートメニューからPythonを起動できるようになります。

インストーラで特にフォルダを指定しない場合、Pythonは「C:¥Python25」にインストールされます。最後の数字はPythonバージョンを表します。バージョンによって違うフォルダにインストールされるわけです。

MacOS XにPythonをインストールする

Mac OS X 10.2以上では、Pythonがあらかじめインストールされています。ただし、Mac OS Xのバージョンによっては2.2など古いバージョンのPythonがインストールされていることがあります。

表：Mac OS XとPythonのバージョン対応

Mac OSのバージョン	Pythonのバージョン
Mac OS X 10.2 (Jaguar)	Python 2.2
Mac OS X 10.3 (Panther)	Python 2.3
Mac OS X 10.4 (Tiger)	Python 2.3

バージョンが古い場合、インストーラを使って最新のPythonをインストールできます。同じく「Python標準リリース」というページに、MacOS X用のインストーラがリンクされています。このインストーラを使って、最新のPythonをインストールしてください。

LinuxにPythonをインストールする

ほとんどのLinuxディストリビューションではPythonがあらかじめインストールされていますが、Pythonのバージョンが2.4より古いことがあるようです。以下のようなコマンドを打って、Pythonのバージョンを確認してください。Vは大文字です。注意してください。

● コマンドラインで確認

```
$ python -V
Python 2.5.1
```

コマンドを打って「Pyton 2.3.4」などと表示された場合は、念のため「python2.5 -V」というようにバージョン番号を補ってコマンドを入力してみ

てください。これでエラーになった場合は、インストールされているPythonのバージョンが2.4より古いということになりますので、Python 2.5を新規にインストールしてください。多くのディストリビューションには**バッケージ管理システム**が搭載されているはずです。バッケージ管理システムを使うと、手軽にPythonをインストールできます。

たとえば「Fedora」というLinuxディストリビューションでは**yum**というパッケージ管理システムを利用してPythonをインストールします。yumを使ってPython 2.5をインストールするには、以下のようなコマンドを打ちます。なお、コマンドを入力する前にスーパーユーザになっている必要があります。

●yumによるインストール

```
# yum install python2.5
```

パッケージ管理システムを使う代わりに、ソースコードを入手してPythonをビルドし、インストールすることもできますが、ここでは詳しく解説しません。

02-02 Pythonの基礎

この節では、簡単にプログラミング言語としてのPythonについておさらいをしたいと思います。

Pythonは、プログラミング言語の中でも**スクリプト言語**、**動的言語**(Dynamic Language)に分類される言語です。同じスクリプト言語の仲間「Ruby」、「Perl」、「PHP」などと並んで**軽量言語**(LL、Lightweight Language)と呼ばれることもあります。

ノリがよく、手軽にプログラムの開発を行えるのがPythonのような軽量言語の特徴です。CやC++、Javaといったプログラミング言語に比べ、少ない時間で多くのプログラムを開発できると言われています。プログラムの修正も容易です。

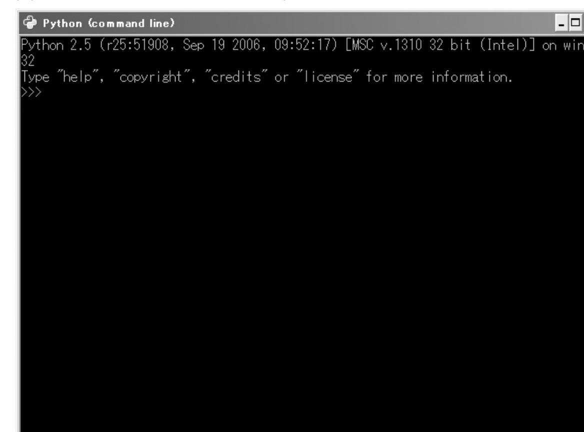
Pythonを使えば、Webアプリケーションで頻繁に扱う文字列データを手軽に扱うことができます。モジュールと呼ばれる機能を活用すれば、いろいろな処理を手軽に行えます。PythonはWebアプリケーションを作るのに向いたプログラミング言語だと言えます。

Pythonの起動

では実際にPythonを使ってみましょう。ここでは**インタラクティブシェル**(対話型シェル)と呼ばれる機能を使ってPythonに触れてみます。インタラクティブシェルを使うと、Pythonを直接操作しながらプログラムを組むことができます。

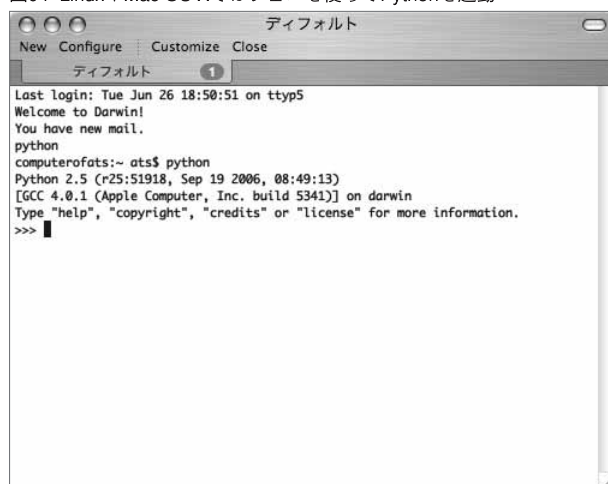
WindowsにPythonをインストールすると、スタートメニューにPythonの項目が登録されます。この中にある【Python (command line)】という項目を選ぶと、Pythonのインタラクティブシェルを起動できます。

図03 Windowsのインタラクティブシェル



LinuxやMacOS Xでは、シェルを使ってPythonを起動します。pythonというコマンドを入力して「Enter」キーを押すと、起動メッセージとともにPythonが起動します。

図04 LinuxやMac OS XではシェルをPythonを起動



インタラクティブシェルを使って、簡単なPythonのプログラムを試してみましょう。>>>という文字列は**プロンプト**と呼ばれています。この文字列は入力する必要がありません。

Pythonを使うと数値だけでなく、ダブルクオーテーションで囲んだ文字列を簡単に扱えます。計算や処理の結果はインタラクティブシェルが自動的に表示しています。

```
>>> 1+1                      # 足し算
2
>>> mach=1225                # 音速(km/h)を変数に代入
>>> topspeed=8600            # サターンVの最高速度
>>> topspeed/mach            # 音速に換算
7
>>> language="Python"        # 文字列を変数に代入
>>> language+" is cool!!"    # 文字列の足し算
Python is cool!!
```

Pythonでは、数値や文字列などを変数に代入することもできます。事前に特別な準備をしないで、変数に数値や文字列を代入していることに注目してください。Pythonでは、数値や文字列などを代入をするだけで変数を自動的に用意してくれます。また、同じ変数に違った種類のデータを代入することができます。このような特徴からPythonは動的言語と呼ばれています。

組み込み型

プログラムで扱うデータにはさまざまな種類があります。数を扱うための数値、テキストを扱うための文字列など、データの種類を**型**と呼びます。Pythonには、プログラムでよく利用するデータを扱うための型が何種類か用意されています。あらかじめ組み込まれていてすぐに利用できる、というような意味で**組み込み(ビルトイン)型**と呼ばれています。

Pythonには以下のような組み込み型が用意されています。

数値型

数を扱うための型です。数の精度に合わせて**整数型、浮動小数点型**に分かれています。四則演算を使って計算を行うことができます。整数と整数の割り算をするときには気をつけてください。Pythonでは計算結果の数値の精度を維持しようとします。そのため、1/2は0.5ではなく0(ゼロ)になります。

数値型の例:

```
123、3.14
```

文字列型

テキストデータを扱うための型です。文字列を定義するにはテキストを引用符で囲みます。引用符1つか、または3つで文字列を囲んで文字列型のデータを定義します。ダブルクオーテーション(")、シングルクオーテーション(')どちらも利用できます。

文字列型の例:

```
"Python"、"""Long string"""
```

また、文字列は**シーケンス型**と呼ばれるデータ型の一種です。インデックスを指定して文字列の要素にアクセスしたり、スライスと呼ばれる機能を使って文字列の一部を取り出すことができます。

ユニコード文字列型

Pythonでは、ユニコード文字列を扱うための型が別に用意されています。文字列と同じようにクオーテーションで文字を囲んで定義しますが、その際に先頭に"u"を付けることでユニコード文字列を定義します。Pythonで日本語のようなマルチバイト文字列を扱うときには、ユニコード文字列として扱うとよいでしょう。

ユニコード文字列の例：

```
u"日本語"
```

Pythonでは、ユニコード文字列から**シフトJIS**や**EUC-JP**のような他のエンコードに変換できます。また、他のエンコードを持った文字列からユニコード文字列への変換も行えます。Pythonには**コデックス**（codecs）と呼ばれる仕組みが備わっています。この仕組みを使うと、ユニコードを中心として複数エンコードへ文字列を変換できます。Python 2.4以上では、日本語のエンコードを扱うための**CJK Codecs**が標準搭載されるようになりました。このため、2.4以上のPythonでは日本語の文字列を扱いやすくなっています。

リスト

複数のデータを順番に並べて管理するために利用する型がリストです。文字列と同じシーケンス型に分類されます。文字列は文字を専門に扱いますが、リストには数値や文字列、そしてリスト自身を含めてあらゆるデータを登録できます。

リストの例：

```
[100, 200, "abc"]
```

順番に並べたデータの一要素は**インデックス**を使って取り出すことができます。インデックスは0から数え始めます。「a_list[2]」のようにすると、a_listというリストの、3番目の要素を取り出せます。「a_list[3]=5」のように要素を指定して代入を行うことで要素の入れ替えができます。リストに対して足し算をして追加したり、メソッドを使って要素の削除ができるのもリストの特徴です。

リストの面白い機能として**スライス**があります。指定した区間にある複数のデータをリストとして取り出す機能です。スライスでは、要素を取り出す範囲をコロン（:）を使って指定します。10個の要素があるa_listというリストに対して「a_list[2:5]」としてスライスを使うと、3番目から5番目の要素を取り出せます。スライスに対して代入を行うと、複数の要素を同時に置き換えることもできます。

辞書（ディクショナリ）

リストと同様に複数の要素を管理するために利用する型です。簡単なデータベースとして利用できます。リストと違うのは、要素を**キー**で管理することです。リストや文字列のようなシーケンス型に対して、辞書のようなデータ型をPythonでは**マップ型**と分類しています。

辞書を定義するときには、キーと対応する値をコロン（:）で区切って定義します。

辞書の例：

```
{'one':1, 'two':2}
```

リストと違い、辞書には順番という考え方がありません。要素を取り出す時には「a_dict['one']」のようにキーを使います。「a_dict['three']=3」のように辞書に代入をするとき、キーがなければ新たなキーが登録されます。キーがある場合には要素だけを入れ替えます。

要素を参照するとき、存在しないキーを指定すると**エラー**（Key Error）になります。キーをスマートに扱うことが、辞書を便利に使いこなすための鍵となります。

タプル型

リストと同様に、複数のデータを順番に管理するために利用するデータ型です。リストと違う点は、要素の入れ替えや削除、追加ができないという点です。変更する必要のないデータ、変更されては困るデータを扱うときに活用します。

タプル型の例：

```
(100, 200, "cde")
```

その他の組み込み型

他にも、Pythonはいくつかの組み込み型を持っています。コンピュータ上のファイルを扱うための**ファイル型**はその1つです。

集合型とも呼ばれる**set型**も組み込み型の一種です。set型はリストのようなシーケンス型のデータ型です。リストとの違いは、要素が重複しないように保たれるという点です。そのため、整数の1という要素を持っているset型のデータにさらに1を追加しようとしても、新しい要素として追加されません。

なお、Pythonでは文字列のパターン検索を行う**正規表現**は組み込みの機能として用意されていません。Pythonで正規表現を使うには、モジュールをインポートする必要があります。

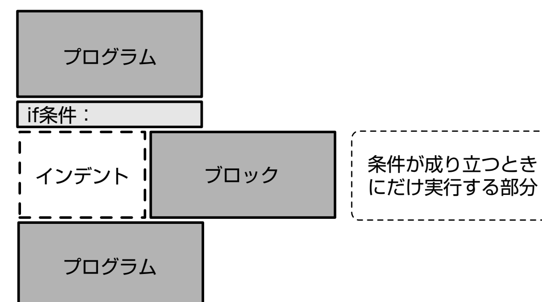
フロー制御

プログラムには、実行したい処理を実行したい順番に記述していきます。処理の流れのことをフローと呼びます。条件によって処理を分けたり、繰り返しを行うことを**フロー制御**と呼びます。

フロー制御では、対象となるコードの「ここからここまで」という範囲を指示する必要があります。プログラムの中で、どこからどこまでを条件によって実行したいのか。またはどの部分が関数に相当するのかを、プログラミング言語に分かるように指示する必要があるわけです。このような部分は**ブロック**と呼ばれています。

Pythonでは、**ブロックを表現するのにインデント**（**字下げ**）**を利用**します。ブロック表現にインデントを使うということが、Pythonの大きな特徴の1つとなっています。ブロックの左側が揃うため、見た目でブロックを判断しやすくなります。このため、自然と読みやすいプログラムを書けるわけです。

図05 Pythonではブロックをインデントする



条件分岐（if文）

Pythonで条件分岐を行うには**if文**を使います。if文には、括弧などを添えず条件式を記述します。条件が**真**（True）のときに実行したいプログラムはインデントします。

if文の例：

```
if year > 2000 and year<=2100:
    print "21st century"
elif year>2100 and year<=2200:
    print "22nd century"
else:
    print "20th century(maybe)"
```

上記の例では数値の入った変数を使って条件分岐をしています。**and**や**or**と組み合わせると、論理式を使った条件分岐ができます。

elifの条件は、最初の条件が成り立たなかったときに評価されます。Pythonにはswitch文がありません。代わりにifとelifを組み合わせるとよいでしょう。**else**のブロックは、どの条件も成り立たなかったときに実行します。

if文に添える条件式には、**==**や**<**のように真偽値を返す比較演算子を使います。Pythonでは、空の文字列や空のリスト、空の辞書などが**偽**（False）として扱われます。要素のある文字列やリスト、辞書は真として扱われます。この性質を使うと、オブジェクトが空かどうかを手早く調べることができます。

シーケンスを使った繰り返し（for文）

Pythonで繰り返しをしたいときに利用するのが**for文**です。Pythonのfor文では、リストのようなシーケンスを**in**の後に添えて記述します。シーケンスの内容を1つずつ取り出し、inの前にある**繰り返し変数**に代入しながらブロックの内部を処理していきます。この例では、組み込み関数range()を使って0から9までのリストを生成して、ループブロック内のコードを実行します。

Pythonではブロックごとに**名前空間**が区切られません。そのため、for文で定義した繰り返し変数や、ループブロックで定義した変数はfor文の後でも利用できます。

for文の例：

```
for cnt in range(10):       # 0から9までの数を表示する
    print cnt
```

条件を使った繰り返し（while文）

条件が成り立つときにだけループを実行したい場合に**while文**を使います。while文には真偽値を返す式を添え、式が真の間だけ、ブロック内のコードを実行します。

while文の例：

```
while cnt < 10:
    print cnt
    cnt+=1
```

関数定義と呼び出し

関数は、繰り返し使う処理をあらかじめ定義しておくために利用します。**def**に続けて関数名と括弧を記述します。関数として実行するプログラムはインデントして記述します。

関数定義の例：

```
def a_function(arg1, arg2):
    """2つの数値をかけ算して返す"""
    return arg1*arg2
```

def文のすぐ下にある文字列は**ドキュメンテーション文字列**と呼ばれています。Pythonには、関数の説明をドキュメンテーション文字列に書いておく、という決め事があるのです。

Pythonでは、関数は**呼び出し可能オブジェクト**とも呼ばれています。Pythonで関数を定義すると、関数として呼び出すことができる「変数のようなもの（=オブジェクト）」が作られる、と思ってください。

事前に定義されていない変数を使おうとするとエラーになります。関数を利用するプログラムの前で、関数定義をしておく必要があります。Pythonでは、関数は変数と同様に扱われるわけです。ですので、別の変数に関数を代入することもできます。

引数の定義

関数にデータを渡して処理をしたいときには、関数に**引数**を定義しておきます。先ほどの例では「arg1」と「arg2」が引数に当たります。「arg1=10」のように、引数名に代入をするような記述をすると、引数のデフォルト値を指定できます。デフォルト値が設定された引数は、呼び出し時に省略することができます。

paramのように、アスタリスク（）を使った引数は特殊な意味を持ちます。定義されている数以上の引数を渡して関数を呼び出すと、あふれた引数がリストとしてparam引数に入って呼び出されます。引数の数を指定しない関数を作りたいときに利用すると便利です。

argsのようにアスタリスクを2つ付けた引数を定義することもできます。このような引数を定義すると、関数は不定個のキーワード引数**を受け取れるようになります。argsには辞書が渡されます。関数内では、キーワード引数の引数名のキーに、値が登録された状態で利用できます。

関数の戻り値とアンパック代入

関数の外部に処理結果を渡したいときには**return文**を使います。Pythonのreturn文には括弧を添える必要はありません。関数の外に受け渡したいデータをそのまま記述します。先ほどの例ではarg1とarg2をかけ算した値を戻り値としています。

Pythonでは、複数のデータを一度に戻り値として返すことができます。

「return a, b, c」のように複数の値をカンマで区切るか、タプルを返します。
関数を呼び出す側では「a, b, c=somefunc()」のように複数の値を受け取ることができます。タプルを自動的に展開して代入することから、これを**アンパック代入**と呼んでいます。

関数と名前空間

Pythonでは代入を行うことで変数を定義します。変数は、どこで定義されたかによって利用範囲を制限されます。Pythonの関数内では、特別な**名前空間**（Name Space）が作られます。関数内で定義された変数は、この名前空間の中でだけ利用できます。ですので、関数で定義した変数を外部で利用することはできません。引数も、関数内で定義された変数と同じような扱いを受けます。
関数の処理結果を外部で利用したいときには、戻り値として関数の外部に値を受け渡します。

関数の呼び出し

事前に定義された関数は、次のようにして呼び出せます。

```
a_function (1, 2)
```

関数が戻り値を返すようになっている場合は、戻り値を変数に代入したり、計算など別の処理に利用できます。戻り値がない場合は**None**という特別な値が返されます。
引数の指定方法には2つの方法があります。上記のように直接値を指定する方法が1つ。そして次のように変数名を**キーワード指定**する方法です。

```
a_function (arg1=1, arg2=2>
```

キーワード指定をすると、定義順に関係なく、引数を指定できます。

モジュールの活用

Pythonには**標準ライブラリ**と呼ばれるものが搭載されます。標準ライブラリを使うことでPythonの機能を大幅に拡張することができます。テキスト処理やネットワーク処理、数値計算など、便利で強力な機能を備えたライブラリがたくさん用意されています。Webアプリケーションなどを作るときに利用する処理の多くが、標準ライブラリにすでに入っています。
Pythonを使えば、基本的な処理を実行する関数をゼロから作る必要はありません。また、インターネットを探し回ってライブラリをインストールする必要もないのです。Pythonをインストールするだけでこのような機能が利用できるため、Pythonは**バッテリ付属（Battery included）言語**と呼ばれることもあります。

モジュールのインポート

モジュールを利用し、Pythonの機能を拡張するには**import文**を使ってモジュールをインポートします。インポートには2つの種類があります。モジュール名を指定してモジュールをインポートする方法と、**from文**を使ってモジュールの一部をインポートする方法です。

モジュールインポートの例：

```
import re
from urllib import urlopen
```

モジュールのインポートは、プログラムのほとんどすべての場所で実行できます。関数やif文などのブロックでもモジュールのインポートを実行できます。
インポートしたモジュールは、変数や関数と同じようにオブジェクトとして扱われます。変数や関数と同様に、モジュールを利用する前にインポートしておく必要があります。また、関数内部のように、特別な名前空間を持っている場所でインポートしたモジュールは、その名前空間の内部でしか利用できません。

モジュールの作成

Pythonのようなプログラミング言語を使うとき、プログラムを事前にファイルに書いておくことがあります。プログラムを書いたファイルを指定してPythonを実行することで、何度もプログラムを実行できます。プログラムを

書いたファイルを**スクリプトファイル**と呼ぶことがあります。

Pythonではスクリプトファイルがモジュールとして扱われます。モジュールを作って関数などを利用したいときには、スクリプトファイルを作ればよいわけです。スクリプトファイルのファイル名がそのままモジュール名になります。

たとえば、a_module.pyというファイルにsome_func()という関数を定義したとします。外部からは、以下のように利用できます。

```
import a_module                  # モジュールをimportする場合
a_module.some_func()

from a_module import some_func   # 関数だけをimportする場合
some_func()
```

モジュールをインポートすると、Pythonはスクリプトファイルを先頭から読み込んで書かれているプログラムを実行します。スクリプトファイルを読み込む課程で、定義されている関数などをモジュールの「持ち物」として登録します。関数だけでなく、変数などもモジュールの持ち物として登録されます。

インデントの一番浅いところを**トップレベルのブロック**と呼びます。トップレベルのブロックに記述されているプログラムは、モジュールをインポートする課程で**必ず実行**されます。モジュールの初期化用のコードは、トップレベルに記述するとよいでしょう。

02-03 Pythonのオブジェクト指向機能

Pythonはオブジェクト指向のスクリプト言語です。新しくクラスを定義したり、既存のクラスを継承する、といったオブジェクト指向言語としての機能を一通り備えています。

Pythonのオブジェクト指向機能はとてもシンプルです。そのようなこともあり、C++やJavaのような他のオブジェクト指向言語を使った経験のある人から見ると、Pythonのオブジェクト指向機能は物足りなく映るかもしれません。しかし、言語全体としての実装はとても見通しが良く作られていますし、必要な機能は備えています。開発を進めるに当たって困ることはほとんどありません。

そもそも、Pythonはオブジェクト指向機能の利用を強制しないような作りになっています。無理にオブジェクト指向機能を使おうとせず、必要に応じて利用するのがPythonの流儀です。

すべてがオブジェクト

純粋オブジェクト指向言語としてPythonを見た場合、その特徴を一言で表すとするなら**すべてがオブジェクト**ということになるでしょう。文字列やリストなどの組み込み型だけでなく、モジュールやクラス、関数やメソッドまでもがオブジェクトとして扱えます。

ありとあらゆるものがオブジェクトなので、関数やクラスを変数に代入したり、挙動を動的に書き換えることができます。この性質をうまく利用すると、複雑なプログラムを驚くほど簡潔に書くことができます。ここでは、一般的によく利用されるPythonのオブジェクト指向機能、**クラス**の定義と利用方法に限って解説します。

クラスを定義する

Pythonでクラスを定義するには**class文**を利用します。class文の後にクラス名を添えて、クラスを定義します。既存のクラスを継承するには、クラス名の後に括弧を添えて継承したいクラスを記述します。多重継承をする場合には、複数のクラスをカンマ(,)で区切って並べます。Pythonでは、組み込み型を含むすべての型やクラスを継承できます。

クラス定義の例：

```
class SomeClass:
    def __init__(self, num):
        """初期化メソッド"""
        self.num=num

    def some_method(self, param):
        """メンバ変数と引数をかけ算して返す"""
        return self. num*param
```

クラスのコードはインデントして記述します。クラスには、**メソッド**や**アトリビュート**（メンバ変数）を定義できます。

Pythonのオブジェクト指向機能はとてもシンプルに作られています。クラス自体と、クラスから生成するインスタンスオブジェクトが独自の名前空間のようなものを持っていて、メソッドやアトリビュートはその名前空間に付属する形で保存されています。

Pythonではプライベート・メソッドを定義することができません。クラスのメソッドはすべて**バーチャルな公開メソッド**となります。クラスのメソッドなどを外部から隠蔽するには、クラス名の前にアンダースコア（_）を1つか、または2つ付けます。

クラスインスタンスの生成

クラスからオブジェクトを生成するには、次のようにクラス名自体を関数のように呼び出します。

```
ins=SomeClass()
```

このようにすると、あらかじめ定義されたクラスからオブジェクトが作られます。クラスから作られるオブジェクトを**クラスインスタンス**と呼びます。**__init__()**というメソッドは初期化メソッドです。クラスのインスタンス生成時に呼ばれる、特殊な働きを持ったメソッドです。Pythonには、デフォルトとなるアトリビュートを定義する方法がありません。このため、たいていは初期化メソッド内でクラスのメンバ変数の初期値を定義します。

Pythonでは、オブジェクトの持ち物にアクセスするためにドット（.）を使います。生成したインスタンスオブジェクトを使ってメソッドを呼び出すには、次のようにします。

```
ins.some_method(10)
```

第1引数self

Pythonのメソッドには、第1引数として**self**を定義しておきます。この引数には、クラスのインスタンスオブジェクト自体が暗黙に渡されます。クラスインスタンス自体には、オブジェクトのアトリビュートやメソッドが持ち物として紐づけられています。メソッド内部では、第1引数selfを使ってクラスインスタンスの持っているアトリビュートにアクセスするわけです。

Pythonでは代入をすると変数が定義されるのと同じように、インスタンスオブジェクトのアトリビュートに代入をすると新しいアトリビュートが作られます。初期化メソッドであらかじめ定義しておきたいアトリビュートに代入を行い、アトリビュートの初期化を行えばよいわけです。

特殊メソッド

Pythonのクラスには、アンダースコア2つで始まるメソッドが何種類か利用できます。このようなメソッドは**特殊メソッド**と呼ばれています。先に解説した初期化メソッド__init__()も特殊メソッドの一種です。

特殊メソッドを使うと、演算子やシーケンスのインデックや辞書型のキーを使ったアクセス、アトリビュートへのアクセスといった言語の基本的な機能をオーバーライドできます。

02-04　ユニコードと日本語処理

前述のとおり、Pythonには2種類の文字列型があります。1つは文字列型、もう1つはユニコード型です。日本語のようなマルチバイト文字列をPythonで扱う場合には、ユニコード型を使うとよいでしょう。

Pythonには**コデックス**（codecs）と呼ばれる仕組みがあります。コデックスを使うと、ユニコードと他エンコードで相互に変換ができます。ユニコード文字列からEUC-JPなどの8ビット文字列への変換もできますし、逆にシフトJIS相当の8ビット文字列からユニコード文字列への変換を行うこともできます。

ユニコード型と文字列の境界

ユニコード型の一番の利点は、マルチバイト文字列の**文字の区切り**を容易に扱えるということです。ASCII文字列でも日本語のマルチバイト文字列でも、1文字ずつ区切って扱うことができます。ユニコード文字列もシーケン

ス型の一種ですので、インデックス指定やスライスを使って文字列の一部を取り出すことができます。以下に簡単な例を示します。

● ユニコード型の使用例

```
>>> u=u"あいうabc"        # ユニコード文字列を変数に代入
>>> print u[2]           # 3番目の文字を取り出す
う
>>> print u[1:4]         # 2番目から4番目までの文字列を取り出す
いうa
```

同じことを、ユニコード相当の8ビット文字列で行うとどうなるでしょうか。

● 文字列型の使用例

```
>>> ru="あいうabc"
>>> print ru[2]

>>> print ru[1:4]
 ･い
```

先の例と同じく2番目に当たる文字列を取得しようとすると、バイト列で見たときの3番目の文字列、ユニコードの内部表現「あ」の3バイト目である文字コード130（16進数で82）に相当する文字列が返ってきます。

このように、8ビット文字列にインデックスを与えても、マルチバイト文字列の文字の区切りを正しく扱えないのです。マルチバイト文字列を簡易に正しく扱いたい場合は、コード内部でユニコード文字列を使うと処理が簡潔に書けます。

他のエンコードからユニコード文字列への変換

Webからの入力や、Eメールの本文などはユニコード以外のエンコードで送られてくることがあります。そのような場合には、文字列をユニコード文字列に変換しておくと便利です。

特定のエンコードを持った8ビット文字列からユニコード文字列に変換するには、2つの方法があります。ここでは、**unicode()関数**を使った方法を紹介します。

unicode()関数の使用例：

```
u=unicode('あいう', 'euc-jp', 'replace')
```

第1引数に変換対象となる8ビット文字列を、第2引数に文字列のエンコードを渡します。すると、ユニコード文字列が戻り値として返ってきます。第3引数はオプションで、変換中に起こったエラーに対する対処を指定します。ここでは、変換不能な文字列があった場合には特定の文字列に置き換えるように指定しています。

なお、次のdecode()メソッドを使っても、特定のエンコードをユニコード文字列に変換できます。

ユニコード文字列から他エンコードへの変換

decode()メソッドの使用例：

```
u=u"あいう"
u.decode('iso-2022-jp', 'ignore')
```

ユニコード文字列を他のエンコードに変換したい場合には、ユニコード文字列に対して**decode()メソッド**を使います。メソッド呼び出しを行っているユニコード文字列が処理の対象になるので、第1引数は変換したいエンコード名となります。第2引数は、変換エラーが起こったときの対処を指示します。この例では、変換エラーが起こった文字列をそのまま残すように指示しています。

プログラム内部での日本語の扱い

Pythonのプログラムでは、内部でユニコードを使い、必要に応じてエンコードを変換するようにします。たとえば、携帯電話向けのWebアプリケーションをPythonで作りたいとしましょう。プログラム内部のユニコード文字列を使い、Webアプリケーションの出力時にシフトJISに変換をするとよいでしょう。また、PythonからEメールを送信するときも、同様に送信前にエンコード変換するようにします。

CHAPTER 03

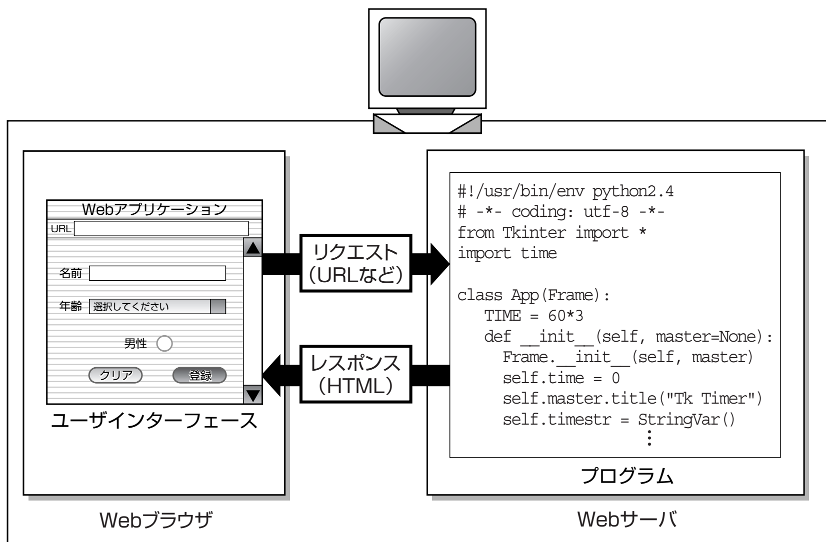
PythonでWebサーバを作る

ここからはいよいよ、Pythonを使ってWebアプリケーションを作っていきます。

Webアプリケーションはクライアント・サーバ型のアプリケーションです。Webアプリケーションを実際に動かすには、クライアントとなるWebブラウザと、サーバとなるWebサーバが必要です。

Webブラウザは、皆さんが日常的に利用しているはずです。Webサーバは、読者の多くが使っていないと思いますので、別途用意する必要があります。本書では、Pythonに含まれる標準ライブラリを使ってWebサーバを用意します。Webサーバを走らせるためには、普段使っているパソコンを使います。Pythonをインストールしたパソコンが1台あれば、Webアプリケーションを作ることができるわけです。

図01 1台のパソコンでWebサーバとクライアントを動かす



Pythonを使うと、驚くほど簡単にWebサーバを動かすことができます。Webサーバ上にPythonのプログラムを設置して、シンプルなWebアプリケーションを作ることもできます。ここでは、Pythonを使ってWebサーバを作る方法について解説します。

03-01 SimpleHTTPServerを使う

Pythonに付属している標準ライブラリにはたくさんのモジュールやクラスが含まれています。中には、シンプルなWebサーバとして機能するモジュールもあります。本来は、このモジュールを拡張してより高機能で実用性のあるWebサーバを作るために用意されているものですが、基本的な機能は一通り備えています。

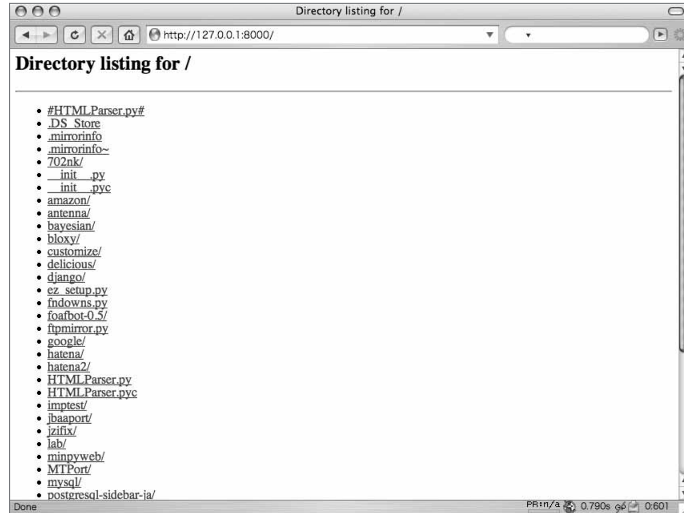
まずはSimpleHTTPServerというモジュールを使って簡単なWebサーバを作ってみましょう。方法はとても簡単で、たった2行のコードを書くだけです。P.23にある方法を使って、インタラクティブシェルを立ち上げてください。インタラクティブシェルを使い、Webサーバを動かしてみましょう。2行目でサーバを起動した後は、Webサーバの稼働状態がメッセージとして表示されます。

● Webサーバの実行

```
>>> import SimpleHTTPServer
>>> SimpleHTTPServer.test()
Serving HTTP on 0.0.0.0 port 8000 ...
localhost - - [18/Jun/2007 15:42:06] "GET / HTTP/1.1" 200 -
localhost - - [18/Jun/2007 15:42:06] code 404、 message File not found
```

この状態で、Webブラウザを使って「http://127.0.0.1:8000/」というURLにアクセスしてみてください。状況によってWebブラウザに表示される内容は違いますが、たいていはファイルのリストが表示されるはずです。

図02 Pythonを使うと2行でWebサーバが起動できる



127.0.0.1というのは**ループバックアドレス**と呼ばれるIPアドレスです。Webブラウザと同じマシンを指しています。コロン(:)の後ろはポート番号です。ブラウザでアクセスしているURLは、「Webブラウザと同じマシンのポート8000番で通信を受け付けているWebサーバにアクセスする」という意味を持っています。

なお、Webサーバを止めるには、LinuxやMac OS XではCtrl+Cを押します。Windowsの場合は本来Ctrl+Zで止まるはずですが、Python 2.5のコマンドラインではうまく終了できないことがあるようです。終了できない場合はウィンドウを閉じてください。

HTMLファイルを扱う

SimpleHTTPServerモジュールを使ったWebサーバでは、起動したときのカレントフォルダ（ディレクトリ）にあるファイルを見せるような動きをします。先ほどWebブラウザに表示されていたファイルのリストは、Pythonのインタラクティブシェルを起動したフォルダ（ディレクトリ）にあったファイルだったわけです。このように、Webサーバがファイルを扱う起点となるフォルダのことを**ドキュメントルート**と呼ぶことがあります。

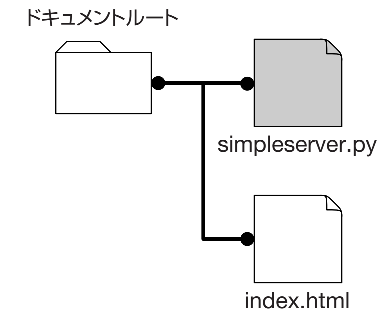
ドキュメントルートに相当する場所に**index.html**という名前のHTMLファイルを置くと、Webブラウザにはそのファイルが表示されます。

Webサーバを起動するスクリプト

今度は、ドキュメントルートに相当するディレクトリを作って、Webサーバを起動するスクリプトとHTMLファイルを置いてみましょう。

まずはフォルダ（またはディレクトリ）を作ります。場所はどこでも構いません。フォルダを作ったら、次のような内容の2つのファイルを置いてみてください。simpleserver.pyはWebサーバを起動するためのPythonのスクリプトです。index.htmlはWebブラウザに表示するHTMLファイルです。

図03 ドキュメントルートに相当するフォルダを作り、ファイルを置く



ファイルはメモ帳のようなテキストエディタで編集して作ります。ワープロなどは使わないでください。

List01 simpleserver.py

```
import SimpleHTTPServer
SimpleHTTPServer.test ()
```

List02 index.html

```
<html><body>
Python is awesome !
</body></html>
```

ファイルを設置したら、Webサーバを起動します。Windowsなら、simpleserver.pyというスクリプトファイルををダブルクリックします。LinuxやMac OS Xでは、シェルを使ってスクリプトファイルを設置したディレクトリに移動し、次のようにコマンドを入力します。

```
$ python simpleserver.py
```

すると、Webサーバが立ち上がります。この状態で、Webブラウザを使って「http://127.0.0.1:8000/」という、先ほどと同じURLにアクセスします。すると、index.htmlの内容が表示されるはずです。

同じように、HTMLファイルやフォルダ、画像やCSSを追加していくと、より複雑なWebページを作ることができます。

03-02 CGIHTTPServerを使う

標準ライブラリには、Pythonで書かれたプログラムをWebサーバの中で実行できるライブラリもあります。それが「CGIHTTPServer」です。CGIHTTPServerを使うと、Webサーバを用意したり設定ファイルを編集することなしに、PythonでWebアプリケーションを作ることができます。Pythonがインストールされたパソコンだけがあればよいので、とても手軽にWebアプリケーションを作って試すことができます。

CGIHTTPServerを起動するスクリプト

先ほどスクリプトファイルとHTMLファイルを設置したフォルダに、別のスクリプトファイルを設置してください。

List03 cgiserver.py

```
import CGIHTTPServer
CGIHTTPServer.test()
```

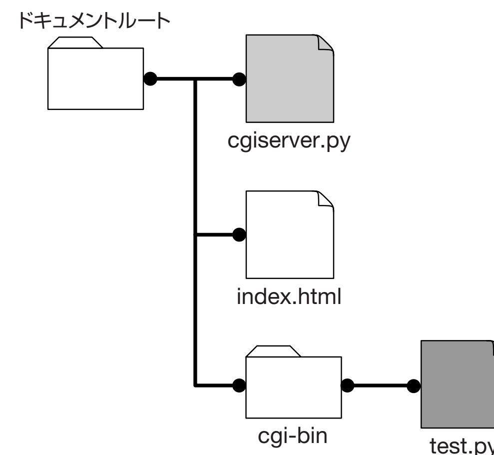
先ほどのスクリプトファイルと同じように、このスクリプトファイルを起動します。すると、同じようにWebサーバが立ち上がります。Webブラウザで**http://127.0.0.1:8000/**というURLを開いてください。index.htmlの内容が表示されるはずです。このままサーバを立ち上げておいてください。

Webサーバにプログラムを設置する

SimpleHTTPServerはファイルや画像を扱う機能しか持っていませんでした。CGIHTTPServerを使うと、サーバ上でPythonのプログラムを動かすことができます。Pythonのプログラムが出力した結果を**レスポンス**として返すことができるのです。レスポンスはWebブラウザが受け取って表示します。つまり、PythonのプログラムとWebブラウザの間で通信ができるわけです。クライアントとサーバの間で通信を行うのは、Webアプリケーションの動作原理の最も基本的な部分です。

では、実際にPythonのプログラムを設置して、Webブラウザに結果を表示してみましょう。cgiserver.pyというスクリプトを設置したフォルダに、「cgi-bin」という名前のディレクトリを作ります。新たに作ったディレクトリの中に、以下のようなPythonのファイルを設置してください。

図04 ドキュメントルートにcgi-binフォルダを作り、プログラムを置く



LinuxやMacOS Xを使っている場合は、ファイルを設置したディレクトリ上で、シェルから「chmod 755 test.py」というコマンドを入力してください。

Webサーバがファイルをプログラムとして実行できるよう、実行権限を与えます。Windowsの場合は、上記のようなパーミッションの変更は必要ありません。

Webブラウザで先ほど開いたURLの最後に、「cgi-bin/test.py」という文字列を追加してください。つまり、Webブラウザから設置したファイルを呼び出す**リクエスト**を出すわけです。すると、スクリプトが出力した内容が、Webブラウザに伝わって表示されているはずです。

List04 test.py

```
#!/usr/bin/env python

print "Content-type: text/html¥n"
print "<html><body>Python is awesome !</body></html>"
```

簡単にプログラムの内容を解説します。このスクリプトは、print文を使って文字列を表示するというとても単純なプログラムです。print文を使って出力した結果が、Webサーバを経由し、ネットワークを通してWebブラウザにレスポンスとして送り返されていることになります。同じような仕組みを使うと、もっと複雑なHTMLをWebブラウザに送ることができます。Webアプリケーションで実行した処理の結果をWebブラウザ上に表示したり、ユーザインターフェースを表示することもできます。

もう1つ、注意してもらいたい点があります。プログラム側では、3行目に**Content-type**から始まる文字列を出力しています。しかし、この文字列はWebブラウザには表示されていません。この部分は**ヘッダ**と呼ばれている部分で、Webブラウザに返すレスポンスには必ず要求されます。

このヘッダの詳細については後ほど解説をします。この時点では、Webアプリケーションを作るときに必ず必要な「おまじない」のようなものだ、と思っていてください。

簡単なプログラムを動かす

test.pyを改造して、もう少し複雑なプログラムを作ってみましょう。現在の日時と時刻を計算して、文字列に変換するプログラムを作ります。文字列をレスポンスとして返すことで、Webブラウザ上に日付を表示できます。簡単な日付表示プログラムを作ってみましょう。

先ほどのtest.pyを以下のように書き換えてみましょう。少し長いプログラムです。注意して入力してください。

List05 test.py（改）

```
#!/usr/bin/env python

import datetime

html_body = """
<html><body>
%d/%d/%d %d:%d:%d
</body></html>"""

now=datetime.datetime.now()

print "Content-type: text/html¥n"
print html_body % (now.year, now.month, now.day,
                   now.hour, now.minute, now.second)
```

このプログラムでは、**datetimeモジュール**を使って現在時刻を取得して表示しています。レスポンスとして返すHTMLを組み立てるためには、Pythonの**文字列フォーマット**という機能を使っています。HTML全体をあらかじめhtml_bodyという変数に定義しておき、最後に現在の日付を埋め込んで出力しています。

プログラムが入力し終わったら、Webブラウザで先ほどと同じURL（http://127.0.0.1:8000/cgi-bin/test.py）にアクセスしてみてください。現在の日付と時刻が表示されたはずです。

プログラムの入力ミスがあると、正しく結果が表示されません。そのようなときは、Webサーバを動かしているシェルを確認してみてください。シェルにエラーの内容が表示されているはずです。

Pythonのエラー表示は**トレースバック**と呼ばれています。一番最後に、エラーの原因となっている箇所が表示されています。注意してプログラムを直してみてください。

● エラー表示の例：コマンドラインで確認

```
localhost - - [19/Jun/2007 12:18:04] "GET /cgi-bin/test.py
HTTP/1.1" 200 -
  File "/Users/ats/dev/python/minpyweb/cgi-bin/test.py", line 12
    prints "Content-type: text/html¥n"
                                      ^
SyntaxError: invalid syntax
```

プログラムがちゃんと動いているなら、Webブラウザに以下のよう現在の日付と時刻が表示されるはずです。

図05 実行結果

もちろん、動かした時間によって表示の内容は変わります。Webブラウザでリロードをすると、Webサーバにリクエストが送信されて、Pythonのプログラムが動き、そのときの時間をレスポンスとして返すわけです。

ところで、現在の日付と時刻を表示するWebアプリケーションを作ってみたわけですが、時計としてみるとちょっと不便です。Webブラウザでリロードをしないと時刻が更新されません。放っておくと、いつまでも古い情報が表示されています。

Webアプリケーションでは、クライアント（Webブラウザ）のリクエストを受けてレスポンスを返します。そのため、動作が受け身になってしまうのです。このことはWebアプリケーションの大きな特徴です。Webアプリケーションを作るときには、このことをよく覚えておく必要があります。

CHAPTER 04

Webアプリケーションに値を渡す

これまで、Pythonを使ってWebアプリケーションの「出力」を行う方法について解説しました。アプリケーションの2つの要素は「出力」と「入力」です。ここでは、もう1つの重要な要素「入力」を行う方法について解説します。

Webアプリケーションでは、Webブラウザにアプリケーションの利用者(ユーザ)が利用するインターフェース(UI)を置きます。Webブラウザから、「リクエスト」という形でWebサーバになんらかの指示を与えます。

Webアプリケーションでは、2つの方法を使ってクライアントからサーバに指示を与えることができます。1つは「GETリクエスト」と呼ばれています。もう1つは「POSTリクエスト」と呼ばれています。ここでは、順を追って2種類のリクエストについて解説します。

04-01　URLを使ってWebサーバに命令を渡す

Webアプリケーションに関する解説をした節で、「URLはリクエストの一種である」と書きました。URLの後半部分は「パス」に分解されます。Webサーバがパスを解釈して、どのようなレスポンスを返すべきかを判断します。パスとして指定した部分にプログラムがある場合は、プログラムの実行結果がレスポンスとして返されます。

URLには、パスの他に**パラメータ**と呼ばれる情報を含めることができます。パスの最後に**クエスチョンマーク**(?)を置くと、その後がリクエストの中で特別に扱われます。

試しに、「http://127.0.0.1:8000/cgi-bin/test.py?foobarbaz」というURLをWebブラウザに入力してみてください。Pythonで作ったWebサーバを立ち上げた状態でURLを入力します。すると、以前と同じようにPythonのプログラ

ムが出力した結果が表示されたはずです。クエスチョンマークの後ろを別の文字に変えても同じ結果になります。URLの中で、クエスチョンマーク以降がパスと別に扱われているわけです。

クエスチョンマーク以降は**クエリ**と呼ばれています。クエリ（Query）とは、「問い合わせ」や「質問」という意味の英語です。リクエストに添える問い合わせ、というような意味ととらえてください。

クエリを使うと、Webサーバ上のプログラムに値を渡すことができます。クエリには、**キーと値のペア**をイコール（=）で挟んだ形でデータを記述できます。複数の値を渡したいときには、キーと値のペアを「アンド（&）」で区切ります。データの構造はPythonの辞書によく似ています。

このようにしてURLに埋め込んだ値は、文字列としてWebサーバ上のプログラムに渡されます。サーバ上のプログラムでクエリを受け取ることで、クライアントからの指示を受け取ることができるのです。

プログラムで値を受け取る

Pythonには、URL上にクエリとして渡された値を受け取るためのモジュールが用意されています。それが**cgiモジュール**です。このモジュールを使うと、クライアントから渡されたリクエストを分割して、Pythonで扱いやすい形に変換できます。

では、実際にURLからクエリを渡し、サーバ上で走るPythonのプログラムで受け取ってみましょう。まずは、以下のプログラムをcgi-binフォルダに置いてください。LinuxやMacOS Xでは、「chmod 755 querytest.py」としてファイルに実行権限を与えるのを忘れないようにしてください。

List01 querytest.py

```
#!/usr/bin/env python

html_body = """
<html><body>
foo = %s
</body></html>"""

import cgi
form=cgi.FieldStorage()                                  ❶

print "Content-type: text/html¥n"
print html_body % form['foo'].value                      ❷
```

このプログラムは、「クエリとして渡されたfooというキーを表示する」という単純なプログラムです。日付表示のプログラムと同じように、Pythonの文字列フォーマット機能を使って結果を表示しています。プログラムの中ほどでは、cgiモジュールをインポートしています。

直後の行で使っているcgiモジュールの**FieldStorage()**という関数がポイントです（❶）。この関数を使うと、クエリとして渡された値をPythonのプログラムで扱えるように変換できます。変換されたオブジェクトは**フィールドストレージ**（FieldStorage）と呼ばれています。フィールドストレージには「form['foo']」のようにクエリのキーを添えて辞書風にアクセスします。辞書風にアクセスして返ってくるオブジェクトからキーの値を得るためには、「value」というアトリビュートを利用します。❷では、「form['foo'].value」のようにしてfooというキーに対応する値を取得しています。

スクリプトを設置したら、PythonのWebサーバを立ち上げた状態で次のURLにアクセスしてください。

http://127.0.0.1:8000/cgi-bin/querytest.py?foo=bar

すると、ブラウザ上に「foo」というクエリに渡した値「bar」という文字列が表示されるはずです。URLの「foo=」の後を書き換えてアクセスしてみてください。Webブラウザに表示される文字列が変わるはずです。

クエリのキーをスマートに扱う

ところで、querytest.pyというプログラムでは、クエリに必ず「foo」というキーがあることを期待するようになっています。試しに、「cgi-bin/querytest.py?bar=bar」のようにしてキーを変えてみてください。エラーが起こるので、Webブラウザには何も表示されないはずです。かわりにWebサーバを立ち上げているシェルにエラーが表示されているはずです。

このプログラムを、もう少し気の利いたものにしたいときはどうすればい

いでしょうか。さまざまな状況で問題なく動くことは良いプログラムの条件です。クエリにキーが存在しなかった場合はエラーになってしまうのでは良いプログラムとは言えません。キーが存在していない場合でも、適切に動くプログラムの方がより良いプログラムと言えます。

組み込み型の辞書型で、キーが存在する場合でも存在しない場合でもスマートに値を取り出したい場合には辞書型のメソッド**get()**を利用します。このメソッドには、第2引数としてキーが存在しなかったときに値として返す「デフォルト値」を設定できます。get()メソッドを使うと、キーの存在を確認する「if文」などを使わずに、辞書型のキーをスマートに扱うことができます。

辞書型のget()メソッドに似たメソッドがフィールドストレージにあります。**getvalue()**というメソッドを使うと、クエリのキーをよりスマートに扱えます。このメソッドには、辞書型のget()と同じように2種類の引数を渡します。1つ目はクエリのキーとなる文字列です。もう1つはオプションの引数で、クエリにキーが存在しなかったときに利用するデフォルト値です。

querytest.pyの最後の行を以下のように書き換えてみてください。クエリを与えないときにもエラーが発生せず、代わりに「N/A」という表示をします。「N/A」というのは「該当なし(Not Applicable)」という意味の略語です。

```
print html_body % form.getvalue('foo', 'N/A')
```

「13日の金曜日」を探すWebアプリケーション

さて、URLを使ってWebサーバで動くプログラムに値を受け渡すことができることが分かりました。ここでは、もう少し複雑な機能を持ったプログラムを作ってみましょう。

西暦を受け取り、その年に「13日の金曜日」がいくつあるかを調べて出力するプログラムを作ってみようと思います。とてもシンプルなプログラムですが、ユーザからの入力に応じて出力を返す機能を持っています。入力と出力があるという意味では、立派なWebアプリケーションと呼べます。

プログラムの機能をまとめると、以下のようになります。

- **西暦はURLにクエリとして埋め込んで渡します**
- **クエリのキーは「year」とします**
- **クエリを受け取って、その年にある「13日の金曜日」を探し出します**
- **結果として、総数と日付を返します**
- **クエリには数字以外の文字列が渡ってくることも考慮します**

cgi-binフォルダに、以下のスクリプトを設置してみてください。条件分岐やループがあり、これまでのサンプルに比べるとかなり本格的なプログラムになっています。

このスクリプトファイルは**UTF-8**で編集、保存してください。LinuxやMac OS Xでは、「chomd 755 find13f.py」として実行権限を与えてください。

プログラムを設置したら、Webブラウザで「cgi-bin/find13f.py?year=3000」というアドレスにアクセスしてみてください。クエリの数字を書き換えると、別の西暦に対する結果を計算して表示します。

List02 find13f.py

```
#!/usr/bin/env python
# coding: utf-8

import cgi
from datetime import datetime

html_body = u"""
<html><head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
</head>
<body>
%s
</body>
</html>"""

content=''

form=cgi.FieldStorage()
year_str=form.getvalue('year', '')
if not year_str.isdigit():                                          ❶
    content=u"西暦を入力してください"
else:
    year=int(year_str)
    friday13=0
```

```
    for month in range(1, 13):
        date=datetime(year, month, 13)                    ❷
        if date.weekday()==4:
            friday13+=1
            content+=u"%d年%d月13日は金曜日です" % (
            year, date.month)
            content+=u"<br />"

    if friday13:
        content+=u"%d年には合計%d個の13日の金曜日があります" % (
        year, friday13)
    else:
        content+=u"%d年には13日の金曜日がありません"

print "Content-type: text/html;charset=utf-8¥n"
print (html_body % content).encode('utf-8')
```

図01 西暦3000年には13日の金曜日が1つだけある

このプログラムでポイントとなるのは2箇所です。

1つ目はクエリに渡された値が数値かどうかを判別する部分です（❶）。クエリに渡される値はすべて文字列型となります。文字列型を数値に変換するには、組み込み関数の**int()**を使います。ただし、int()に数値以外の文字列を与えるとエラーになってしまいます。そこでこのプログラムでは、Pythonの文字列型が持つ**isdigit()**メソッドを使って事前にチェックを行っています。このメソッドは、文字列が数字だけで構成されている場合に真（True）を返します。ですので、クエリ文字列をint()関数の引数として与えて、エラーになるかどうかを検査できるわけです。クエリが数値以外の文字列を含んでいた場合には、警告の表示を、そうでない場合には13日の金曜日を探す処理を進めます。

もう1つのポイントは、クエリとして渡された西暦に「13日の金曜日」が何日あるかを数える部分です（❷）。このプログラムでは、datetimeモジュールを活用しています。1月から12月までの13日に相当するdatetimeオブジェクトを作り、weekday()メソッドで曜日を調べています。月曜日が0となりますので、4に相当する日が金曜日となります。

13日の金曜日が見つかったら、ループの中でレスポンスの一部となる文字列を組み立てていきます。最後にPythonの文字列フォーマットを使ってレスポンス全体となる文字列を組み立て、print文で出力しています。

04-02　フォームの処理

URLにクエリを埋め込み、Webサーバ上で動くプログラムに値を渡す方法は簡易で便利ですが、欠点が多いのも事実です。長い文字列や複雑な値を渡すのには不向きですし、だいいちアプリケーションの利用者にとって扱いやすい方法とは言えません。そこでここでは、「フォーム（Form）」を使ってもう少し気の利いたUIを作ってみましょう。

フォームを使ってWebサーバに値を渡す

先ほど作った13日の金曜日を数えるプログラムでは、クエリとして西暦を渡していました。クエリを書き換えて西暦を指定するのはとても不便です。ここでは、フォームを使って西暦を入力、指定できるようにしてみましょう。つまり、アプリケーションをコントロールするためのUIを作るわけです。

フォームで作るUIでも、西暦を入力するための「入力窓」を設置します。また、他に部品として「ボタン」を設置する必要があります。フォームでは、

ボタンを押して初めてリクエストが送信され、処理の結果をレスポンスとして得ることができます。

次のような内容のHTMLファイルを作って、cgi-binのあるフォルダと同じフォルダに設置します。つまり、プログラムを設置したフォルダの「1つ上」にファイルを設置することになります。ファイルを保存するときの文字コードは「UTF-8」を使ってください。

List03 f13form.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
</head>
<body>
    <form action="/cgi-bin/find13f.py" method="GET">
        13日の金曜日が何日あるかを探します。<br />
        西暦を入力してください :
        <input type="text" name="year" />
        <input type="submit" name="submit" />
    </form>
</body>
</html>
```

HTMLについて簡単に解説します。<form>タグで囲まれた部分が、ユーザインターフェースの定義になっています。<form>タグには2つの「アトリビュート」があります。1つは**action**というアトリビュートで、文字列の形で先ほど作ったプログラムのパスを指定しています。このようにすることで、リクエストを送る先を指定するのです。もう1つは**method**というアトリビュートです。このアトリビュートについては後ほど解説します。

<form>タグの中には、2つの<input>タグがあります。1つは、西暦を入力するための入力窓です。typeというアトリビュートに**text**という文字列を指定して、テキストフィールドを表示しています。**name**というアトリビュートには、クエリのキーが埋め込まれています。このようにすることで、テキストフィールドに埋め込まれた文字列をクエリの値として送信することができるのです。

もう1つの<input>タグではリクエストを送信するためのボタンを表示しています。**type**アトリビュートに**submit**という文字列を指定すると、リクエスト送信用のボタンが表示されます。

フォームを表示するHTMLファイルを設置したら、次のURLを開いてください。

http://127.0.0.1:8000/f13form.html

もちろん、Pythonで作ったWebサーバは起動しておきます。フォームが表示されたでしょうか。西暦に相当する数字を入力して、結果を確かめてみてください。

図02 西暦入力用のフォーム

テキストフィールドに西暦を入力してボタンを押すと、Webサーバにリクエストが送信されます。リクエストを送信する先は13日の金曜日を数えるプログラムです。

ボタンを押したあとWebブラウザのURLを確認してみてください。「/cgi-bin/find13f.py?year=...」のようになっているはずです。つまり、フォームに入力された内容がクエリに変換され、URLに埋め込まれているわけです。プログラムでは、URLに埋め込まれたクエリから情報を受け取り、13日の金曜日を探し出しています。

フォームの動的出力

これまでは、UIとなるフォームをHTMLファイルとして書き出していました。HTMLファイルもWebアプリケーションの出力も、テキストベースのレスポンスであるという意味では同じです。つまり、プログラムを使ってUIを表示する、ということも可能なのです。

HTMLファイルのように、事前に作っておいた静的なファイルをWebサーバからレスポンスとして出力する方法を「静的出力」と呼ぶことがあります。それに対して、プログラムを使ってレスポンスを出力することを「動的出力」と呼ぶことがあります。

静的出力は、HTMLファイルなどをあらかじめ作ってサーバに設置しておけばよいため、とても手軽に行えます。半面、いつも決まった内容しか出力できません。

動的出力は、レスポンスを返すプログラムを作る必要があるため少々面倒です。半面、状況に応じて出力される文字列の内容を変えることができます。Webサーバに表示される内容を、状況に応じて変えることができるわけです。

フォームを作るとき、状況によって表示内容などを変えたいことがあります。たとえば、先ほどの「13日の金曜日」を表示するWebアプリケーションで、西暦をメニュー（<select>タグ）を使って表示するとしましょう。現在や、その付近の西暦について調べたいことが多いでしょうから、以下のようにメニューを表示するのが親切です。

・現在を起点に、過去、未来それぞれ10年を表示
・現在の西暦を選択状態で表示

「現在の西暦」は、Webアプリケーションをいつ動かすかによって変化します。このような場合は、フォーム・コントロールに相当する文字列を動的に作り、Webブラウザに表示した方がよさそうです。

先ほどのプログラムを改良して、結果だけでなくフォームを表示するようにしてみましょう。「find13f.py」という名前でcgi-binの下にファイルを保存し、WebブラウザでプログラムのURLにアクセスしてみてください。プログラムで作ったフォームが表示されるはずです。

List04 f13form.py

```
#!/usr/local/bin/python
# coding: utf-8

import cgi
from datetime import datetime

html_body = u"""
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
</head>
<body>
  <form method="POST" action="/cgi-bin/find13f.py">
    西暦を選んでください:
    <select name="year">
      %s
    </select>
    <input type="submit" />
  </form>
  %s
</body>
</html>"""

options=''
content=''

now=datetime.now()
for y in range(now.year-10, now.year+10):
    if y!=now.year:
        select=''
    else:
        select=' selected="selected"'
    options+="<option%s>%d</option>" % (select, y)

form=cgi.FieldStorage()
year_str=form.getvalue('year', '')
if year_str.isdigit():
    year=int(year_str)
    friday13=0
    for month in range(1, 13):
        date=datetime(year, month, 13)
```

```
        if date.weekday()==4:
            friday13+=1
            content+=u"%d年%d月13日は金曜日です" % (
            year, date.month)
            content+=u"<br />"

    if friday13:
        content+=u"%d年には合計%d個の13日の金曜日があります" % (
        year, friday13)
    else:
        content+=u"%d年には13日の金曜日がありません"

print "Content-type: text/html;charset=utf-8¥n"
print (html_body % (options, content)).encode('utf-8')
```

このプログラムでは、フォーム・コントロールのうちメニューの項目(<option>タグ)に相当する文字列をプログラムで作っています。メニュー項目に相当する文字列はループを使って作っています。ループの中では、項目の西暦が現在の西暦かどうかを判別して、「selected」というアトリビュートを追加しています。

2つのmethod：GETとPOST

先ほどのフォームにあった<form>タグには「method」というアトリビュートがありました。このアトリビュートは、クエリの送信方法を指定するために利用されます。**GET**という文字列を指定すると、クエリをURLに埋め込んで送信します。

methodアトリビュートには、GETの他に**POST**という文字列も指定できます。先ほどのHTMLの「method="GET"」の部分を「method="POST"」に書き換えて、ブラウザで開いてみてください。西暦を入力してボタンを押すと、今度はURLにクエリが「現れなくなった」はずです。

URLにはクエリがありませんが、プログラムは正しく動いています。プログラムでは、yearというクエリがない場合は「西暦を入力してください」という警告表示をするはずです。警告表示がなく、ちゃんと13日の金曜日を計算しているということは、なんらかの方法でクエリが送信されているということになります。このように、POSTメソッドを使うと、URLの一部を使わずにクエリを送信することができるのです。

GETメソッドとPOSTメソッドの違い

GETメソッドは、特定の性質を持ったデータを取得したいときに利用されます。GETメソッドは、Webアプリケーションに送信するデータがURLとして残ります。このため、リンクにクエリを含んだURLを埋め込む、というような使い方ができます。何度も同じ内容のクエリを送信できるのです。半面、クエリの長さに制限があります。そのため、大きなサイズのデータを送ることができません。

一方、POSTメソッドは、データの新規作成や更新のために利用されます。データのサイズにも制限がないため、大きなデータを送信できます。ただし、クエリのデータはリクエストの背後で送信されます。同じ内容のクエリを再送信するには、フォームなどから同じデータを再度POSTする必要があります。

GETとPOSTの違いや、具体的にどのような方法でクエリが送信されるかについては、後ほど詳しく解説します。

いろいろなフォーム・コントロール

Webアプリケーションではフォームを使ってUIを表示します。フォームを使って表示できるUIにはいくつかの種類があります。先ほどの例で使った「テキストフィールド」と「ボタン」はよく使われるUIの例です。このようなUIの部品を**コントロール**と呼びます。

Web上にコントロールを表示するために、いくつかのタグが定義されています。コントロールを表示したい位置にタグを記述することで、Webアプリケーションで利用するUIを表示できるのです。

コントロールを表示するタグには、nameというアトリビュートを置くようにします。このアトリビュートに指定された値が、フォームから送信されるクエリのキーとなります。コントロールの種類によっては、さらに追加のアトリビュートを指定できたり、タグの中に入れ子のタグを記述する必要がある場合があります。

ここでは、フォーム・コントロールの中で最もよく利用されるものをいくつか紹介します。フォームに入力された値をWebアプリケーション側でどのように扱うかについてもあわせて解説します。

図03 いろいろなフォーム・コントロール

テキストフィールド

1行のテキストを入力するために利用するコントロールです。nameアトリビュートにクエリのキーを指定します。sizeアトリビュートを指定すると、コントロールの横幅を指定できます。コントロールにデフォルト文字列を表示したいときには、valueアトリビュートに文字列を指定します。

テキストフィールドの例：

```
<input type="text" name="first_name" size="20" />
```

Webアプリケーション側で受け取るテータは文字列のデータになります。日本語のようなマルチバイト文字列がある場合には注意が必要です。Webアプリケーション側では、文字列データは8ビット文字列として受け取ります。Pythonのプログラムでは、クエリをユニコード文字列に変換をするとよいでしょう。

クエリ上の文字列の扱いについては、のちほど重点的に解説します。

サブミットボタン

フォームからリクエストを送信するボタンとして利用するコントロールです。nameアトリビュートにクエリのキーを指定します。valueアトリビュートに指定した文字列はボタンのラベルとして表示されます。

サブミットボタンの例：

```
<input type="submit" name="submit" value="送信" />
```

同じフォームに複数のサブミットボタンを設置し、name、valueを変えることで、Webアプリケーション側で押されたボタンを判別できるようになります。たとえば、フォームに「add」と「update」というnameを持つ二種類のsubmitボタンを設置するとします。addというnameのボタンを押すと、もう1つのサブミットボタンupdateのクエリは送信されません。Webアプリケーション側では、クエリのキーを調べることによって、どのボタンが押されたかを知ることができるわけです。

リセットボタン

typeアトリビュートを「reset」とすると「リセット」ボタンを設置できます。リセットボタンは、フォームの入力値をクリアするためのボタンです。valueアトリビュートに指定した文字列がボタンのラベルとして表示されます。

リセットボタンの例：

```
<input type="reset" name="reset" value="クリア" />
```

ラジオボタン

複数の項目から1つだけを選択する、というコントロールを作るときにラジオボタンを使います。テキストフィールドなどと同じく<input>タグを使い、typeアトリビュートにradioという文字列を指定します。

ラジオボタンの例：

```
<input type="radio" name="gender" value="male">男性
<input type="radio" name="gender" value="female">女性
```

同じnameを持つラジオボタンのタグがグループとして扱われます。タグにcheckedというアトリビュートを「checked="checked"」のように設置すると、そのコントロールがあらかじめ選択された状態で表示されます。

同じグループのラジオボタンは1つだけ選択することができます。複数の項目から複数選択するようなコントロールが必要なときにはチェックボックスを使います。

Webアプリケーションには、選択されたコントロールのvalueアトリビュートが、nameに対する値として送信されます。受け取るデータ型は文字列です。valueを変えることによって、どのコントロールが選択されているかを知ることができるわけです。

チェックボックス

複数の項目から複数を選択する、というコントロールを作るときにチェックボックスを使います。<input>タグを使い、typeアトリビュートにcheckboxという文字列を指定します。同じnameを持つチェックボックスのタグがグループとして扱われます。複数の項目を選択できる、という点を除いては、ラジオボタンにとても似たコントロールです。コントロールをあらかじめ選択した状態で表示したいときには、ラジオボタンと同様に「checked="checked"」のようなアトリビュートをタグに設置します。

チェックボックスの例：

```
<input type="checkbox" name="language" value="Python">Python
<input type="checkbox" name="language" value="Ruby">Ruby
```

Webアプリケーションには、選択されているコントロールのvalueが、nameに対する値として送信されます。valueを変えることによって、どのコントロールが選択されているかを知ることができるわけです。

項目が複数選択された場合、Webアプリケーション側では「文字列のリスト」としてデータを受け取ります。同じフォーム中に存在する、同じnameアトリビュートを持つコントロールが複数選択された場合には、プログラムではnameに相当するキーの値として文字列のリストを受け取るのです。プログラム側では、あるnameがどのような種類のコントロールから送られてくるのかを、あらかじめ想定して処理をする必要があります。

メニュー

主に複数の項目から1つ選択するために利用するコントロールです。機能としてはラジオボタンと同じですが、コントロールを表示する面積が小さくて済むのが特徴です。半面、すべての選択項目を見るにはメニューをクリックしなければならない、という不便さがあります。複数の選択項目から1つだけ項目を選択する必要があり、選択項目が多い場合に利用するとよいでしょう。

メニューの例：

```
<select name="language">
  <option value="python">Python</option>
  <option value="ruby">Ruby</option>
  <option value="perl">Perl</option>
  <option value="php">PHP</option>
</select>
```

例を見ると、これまでのコントロールとタグの構成が大きく異なっているのが分かると思います。コントロール全体を囲む<select>タグがあり、ここにnameアトリビュートを記述します。メニュー項目の要素は、<select>タグの子供となるように<option>タグを書き指定します。項目名とクエリの値に相当する文字列は、<option>タグの開始タグと終了タグで囲んで指定しています。<option>タグにvalueというアトリビュートを設置すると、項目が選択されていたときに送信されるクエリの文字列を指定できます。selectedというアトリビュートを「selected="selected"」のように設置すると、その項目が選択された状態でコントロールが表示されます。

テキストエリア

複数行にわたる、テキスト入力用のコントロールを表示します。開始タグと終了タグをペアにして記述します。nameアトリビュートでクエリのキーを指定するのは他のコントロールと同じです。また、colsアトリビュートで入力エリアの横幅を、rowsアトリビュートで縦の長さを指定します。

テキストエリアの例：

```
<textarea name="body_text"
 cols="40" rows="20" >文章を入力してください</textarea>
```

タグで囲まれた部分はデフォルトの文字列としてコントロールの中に表示されます。開始タグと終了タグの間に改行やスペースがあると、文字列としてコントロールの中に表示されます。コントロールの中を空にしておきたい場合には、開始タグと終了タグの間をあけないようにしておきます。

Webアプリケーション側で受け取るデータは文字列となります。Pythonのプログラムでは、必要に応じてユニコード型に変換して処理をします。

04-03 データ型の変換

フォームを含み、Webアプリケーションの受け取るクエリのデータはすべて文字列です。数字を入力するフィールドであっても、Pythonで受け取るデータは文字列となります。

Pythonは、Perlなどと比べ型の扱いが厳密です。そのため、Webアプリケーションで扱うデータは明示的に**型の変換**を行う必要があります。

たとえば、数字のみで構成された文字列型のテータと数値型で足し算をしようとするとエラーになります。インタラクティブシェルを使って試してみましょう。

```
>>> "123"+456
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Pythonでは、演算をする前に型を揃えてやる必要があります。数字だけで構成される文字列を整数に変換するには、組み込み関数の「int()」を使います。文字列として連結したい場合には、組み込み関数の「str()」を使います。

```
>>> int("123")+456
579
>>> "123"+str(456)
123456
```

勝手に変換される方が便利、という考え方があるかもしれませんが、それはPythonの思想に反します。あいまいさをできるだけ排除する、というのがPythonの思想なのです。数値を文字列として扱って連結するのか、または文字列を数値として扱って足し算するのか、型を揃えることで明確に処理の内容を指示し、誰が読んでも分かりやすいプログラムを書く、というのがPythonの思想なのです。

Webアプリケーションに送られてくるほとんどのデータは文字列です。Pythonはデータの型を厳密に扱う傾向にあります。このため、処理の内容によっては型の変換を行う必要があります。

フォームのどの部品にどの値が入力されたかを知るには、コントロールのnameを使います。クエリを取り出し処理を行うときに、nameを指定してデータを取り出し、扱うデータの性質によって文字列型から変換を行うわけです。

データの変換で最もよく使われるのは文字列型から数値型への変換です。Pythonの数値型には、整数型と浮動小数点型があり、変換の方法も別になります。

文字列型から整数型への変換を行う

組み込み関数の**int()**を使います。引数として数字、符号のみを含む文字列を与えると、数値型が戻り値として返ってきます。int()には、小数点や数字以外の文字を含まない文字列を与える必要があります。数字以外の文字列をint()に与えるとエラー(ValueError)が発生します。

文字列型から浮動小数点型への変換を行う

組み込み関数の**float()**を使います。引数として数字、符号、小数点を含む文字列を与えると、浮動小数点型のデータが返ってきます。int()と同じく、英字など余計な文字列を含む文字列を与えるとエラー(ValueError)が発生します。

数値型に変換できるかどうかを調べる

int()、float()とも、変換の妨げになるアルファベットのような文字を含む文字列を引数として与えるとエラーが発生します。そのため、処理を行う前に文字列の内容を確認する必要があります。このような処理を「文字種の検査」と呼ぶことがあります。

文字種の検査は正規表現などを使っても可能です。しかし、文字列が数値のみで構成されているかどうかを調べたい場合には、文字列メソッドの**isdigit()**を使うと便利です。このメソッドは、文字列が数字のみで構成されている場合に真(True)を返します。

```
>>> "1234".isdigit()
True
>>> "123F".isdigit()
False
```

isdigit()は整数であることを検査するためには便利です。しかし、プラス、マイナスなどの符号、小数点が含まれていても偽（False）が返ってきます。このため、浮動小数点などを含む数値の検査には利用できません。

符号を含んだ文字列、浮動小数点に変換する文字列を検査する場合には、別の方法を使うことになります。正規表現を使った方法も1つのアイデアですが、ここでは「例外（エラー）」を使って、スマートに型のチェックを行う方法について検討してみましょう。与えられた数値によって、適切な型のデータを作って返す関数を作ってみます。文字列にアルファベットなどが交じっていて変換できなかった場合はゼロを返します。

```
def isNumber(num_str):
    """文字列を適切な型の数値に変換する"""
    try:
        value=float(num_str)
        if value==int(value):
            return int(value)
    except ValueError:
        return 0
```

まず、関数をtry～exceptで囲みます。この中で起こった例外（エラー）のうち、ValueErrorのみを捕まえます。例外が起こったときにはゼロを返します。引数の文字列に、変換の妨げになる文字列が混じっていた場合、float()関数を呼んだときに例外が発生します。

その後、int()関数を使って再度変換を試みています。文字列に浮動小数点が含まれていないときには、整数を返すわけです。

ユニコード型への変換

WebアプリケーションのUIから、クエリとして送られてくるデータは文字列です。日本語のようなマルチバイト文字列は、特定エンコードの8ビット文字列となって送られてきます。Pythonで日本語のようなマルチバイト文字列を扱う場合には、ユニコード文字列を使うのがなにかと便利です。このため、日本語のようなマルチバイト文字列が含まれる可能性のあるクエリを処理する場合は、8ビット文字列をユニコードに変換することになります。

実際にどのような手法を使ってユニコード型への変換を行うかを説明する前に、簡単にエンコードについて解説したいと思います。

コンピュータで扱う文字には、すべて番号が振られています。どの文字にどの番号を振るか、という取り決めのことを**エンコード**（符号化方式）と呼んでいます。アルファベットや数字、よく利用する記号などを含んだASCII文字には、共通したエンコードがあります。

しかし、日本語や中国語、韓国語などアジア圏の国々では、複数のエンコードが存在します。同じ「あ」という文字をコンピュータで表現するために、複数の数字が割り当てられている、ということです。コンピュータ上に日本語のようなマルチバイト文字を表示するとき、エンコード方式を間違えると正しい文字列が表示されません。一般的に言われる「文字化け」はこのように起こります。

フォームなどから送られてくるクエリをユニコード文字列に変換するためには、このエンコードの情報が必要です。ユニコード文字列に変換を行う関数にエンコード名を文字列として与えることで、正しくエンコードの変換が行えます。

特殊なケースを除いて、フォームから送信される文字列のエンコードは、フォームなどを表示しているWebページのエンコードと同じと思ってほぼ間違いありません。フォームのHTML自体のエンコードはあらかじめ分かっていることが多いわけですから、特別な場合を除いてエンコードの変換に困ることはないでしょう。

縦書き変換プログラム

ここで、マルチバイト文字列を扱う簡単なWebアプリケーションを作ってみることにしましょう。フォームに入力された文字列を縦書きに変換して表示するプログラムを作ります。改行なし、いわゆる全角文字列のみを想定しています。また、カギ括弧や区点などの記号類を縦書き用に変換しない、という簡易なものを作ります。

文字列を縦書きにするためには、文字を1字ずつ分割して並べ替えを行わなければなりません。Pythonの8ビット文字列では、マルチバイト文字の区切りを識別するために特別な処理が必要です。ユニコード文字列に変換すれば、文字の区切りを正しく扱うことができます。

まずはUIとなるフォーム（verticalize_form.html）を設置します。場所はドキュメントルート（PythonのWebサーバを設置したフォルダ）と同じ位置です。

List05 verticalize_form.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
</head>
<body>
<form action="/cgi-bin/verticalize.py" method="POST">
    <textarea name="body" cols="40" rows="20"></textarea>
    <br />
    <input type="submit" name="submit" />
  </form>
</body>
</html>
```

その後、cgi-binフォルダにPythonのプログラム（verticalize.py）を設置します。実行権限を与えて、プログラムとして動くように設定してください。ファイルの設置が終わったら、PythonのWebサーバを起動して「~/verticalize_form.html」というURLにアクセスします。いわゆる全角文字、改行を含まない文字列を入力し、ボタンを押すと縦書きに変換されて表示します。

List06 verticalize.py

```
#!/usr/bin/env python
# coding: utf-8

import cgi
form=cgi.FieldStorage()

html_body = u"""
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
</head>
<body>
%s
</body>
</html>"""

body_line=[]
body=form.getvalue('body', '')

body=unicode(body, 'utf-8', 'ignore')                                  ❶
for cnt in range(0, len(body), 10):
    line=body[:10]
    line+=''.join([u'□' for i in range(len(line), 10)])                ❷
    body_line.append(line)
    body=body[10:]

body_line_v=[u''.join(reversed(x)) for x in zip(*body_line)]❸

print "Content-type: text/html¥n"
print (html_body % '<br />'.join(body_line_v)).encode('utf-8')
```

このプログラムでは、文字列を「リストのリスト」と見なし、右に90度回転することで縦書きの文章を得る、という方針で処理を進めます。リストのリストに分割する過程で、マルチバイト文字の区切りを正しく判定する必要があります。そのために、プログラムの中ほどでクエリを取り出し、unicode()関数を使ってユニコード文字列に変換しています（❶）。

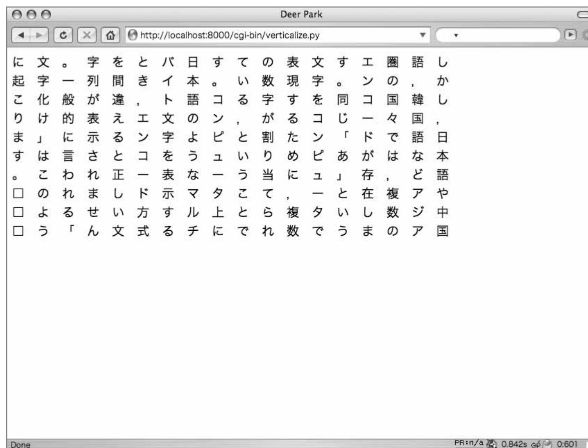
その後のループでは、文字列を10文字ごとに分割して文字列のリストを作っています。10文字より短い行があった場合には、埋め草（□）を使って文字列の長さを揃えています（❷）。

ループの後が、文字列のリストを右90度に回転している部分です。一番のキモは、組み込み関数zip()を使った部分です（❸）。組み込み関数zip()は引数として複数のシーケンスをとり、シーケンスの組み替えを行います。たとえば3×3の多次元配列のようなシーケンスをzip()にかけると、シーケンスの回転（実際には写像）を行うことができます。以下の例が分かりやすいでしょう。なお、この例では改行を調整しています。

```
>>> zip ((1, 2, 3) ,
        (4, 5, 6) ,
        (7, 8, 9))
[ (1, 4, 7) ,
  (2, 5, 8) ,
  (3, 6, 9) ]
```

この性質を使うと、横書きを縦書きに変換することができます。日本語の縦書き文章は右から左に読み進めます。zip()で得た変換後のシーケンスをリスト内包表記で再度展開し、組み込み関数reversed()で左右を反転しています。

図04 縦書き変換Webアプリケーション



04-04 クエリとリスト

Webアプリケーションのuiとなるフォーム・コントロールの中には、複数の値を指定できるものがあります。チェックボックスがその1つです。チェックボックスで複数の項目を選択すると、Pythonのプログラムではリストとして受け取ることになります。また、同じフォームの中に同一のnameを持つコントロールが複数あり、それぞれの項目に値が設定されている場合も、同様にリストが渡ってきます。

ただし、チェックボックスに相当するクエリに必ずリストが返ってくるとは限りません。チェックボックスで1つしか項目が選択されなかった場合には、クエリには文字列が返ってきます。つまり、コントロールの選択状態によってクエリの内容が異なるわけです。

クエリの内容がコントロールの選択状態によって異なると、ちょっと困ったことが起こります。特にPythonの場合、リストも文字列型も同じシーケンス型です。リストが返ってくることを前提にプログラムを書くと、文字列が返ってきたときもリストが返ってきたときも、表面上は問題なく処理が実行できてしまうのです。

簡単なプログラムを作って確認してみましょう。チェックボックスのあるUIと、リクエストを受け、チェックされた項目を表示するプログラムを設置してみます。「checkbox.html」はPythonのWebサーバと同じ階層に、「querytest.py」はcgi-bin以下に実行権限を与えて設置してください。

List07 checkbox.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
</head>
<body>
    <form action="/cgi-bin/querytest2.py"
          method="POST">
        <input type="checkbox" name="language" value="Python"/>
        Python<br />
        <input type="checkbox" name="language" value="Ruby"/>
        Ruby<br />
        <input type="checkbox" name="language" value="Perl"/>
        Perl<br />
        <input type="checkbox" name="language" value="PHP"/>
        PHP<br />
        <input type="submit" name="submit" />
    </form>
</body>
</html>
```

List07 querytest2.py

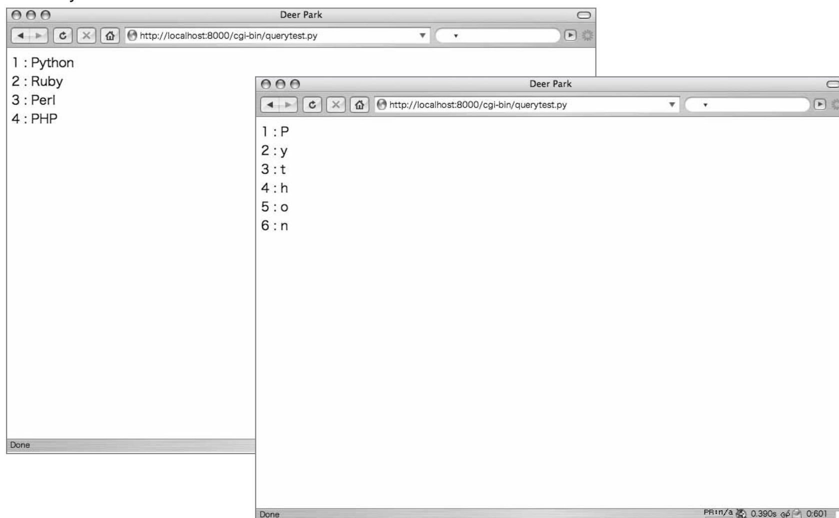
```
#!/usr/bin/env python
# coding: utf-8

import cgi
form=cgi.FieldStorage()

html_body = u"""
<html>
<head>
```

```
<meta http-equiv="content-type" content="text/html;charset=utf-8">
</head>
<body>
%s
</body>
</html>"""

content=''
for cnt, item in enumerate(form.getvalue('language')):
    content+="%d : %s <br />" % (cnt+1, item)

print "Content-type: text/html¥n"
print (html_body % content).encode('utf-8')
```

プログラムでは、組み込み関数**enumerate()**を使ってシーケンスの要素と要素番号を取り出し、ループ処理を行っています。ループの中では、文字列フォーマット機能を使って、結果として表示する文字列を組み立てています。

チェックボックスを2つ以上選択すると、選択した項目が期待通りに表示されます。チェックボックスが1つの場合は、文字列が分割されて表示されてしまいます。文字列もシーケンスなので、ループの中で1文字ずつに分解され、処理されてしまうわけです。

図05 Pythonではリストも文字列も同じシーケンス



このようなことを避けるためには2つの対策が考えられます。

1つは、事前にクエリの型を調べる方法です。組み込み関数の**isinstance()**を使うと、型のチェックが行えます。文字列型であることを調べるためには、**basestring**が利用できます。リスト型であることを調べるためには**list**が利用できます。インタラクティブシェルで試してみましょう。

```
>>> isinstance('123', basestring)
True
>>> isinstance('123', list)
False
>>> isinstance(['Python', 'Ruby'], list)
True
```

このような方法を使い、クエリの型のチェックが実行できます。型チェックを行い、if文などで場合分けをして処理を振り分けるのです。

もっと簡易な方法は、クエリのデータを持っている**FieldStorageオブジェクト**のメソッドを使う、という方法です。FieldStorageオブジェクトには、getvalue()メソッドの他にも、クエリの値を取り出すためのメソッドがいくつか定義されています。

● getfirst (name [, default])

nameを引数に指定して、クエリを取り出します。対象となるnameの値が1つの場合も複数の場合も、必ず最初の要素となる文字列を返します。複数選択することのないコントロールの値を取り出すときに利用すると便利です。

● getlist (name)

nameを指定してクエリの要素を取り出します。戻り値は必ず文字列のリストになります。

ここで紹介したメソッドは、フォーム上にnameが複数あるケースをスマートに扱うように設計されています。先ほどのプログラムでは、必ず文字列のリストを得た方が処理が楽になります。ですので「form.getvalue('language')」の代わりに「form.getlist('language')」とすればよいことになります。

CHAPTER 05

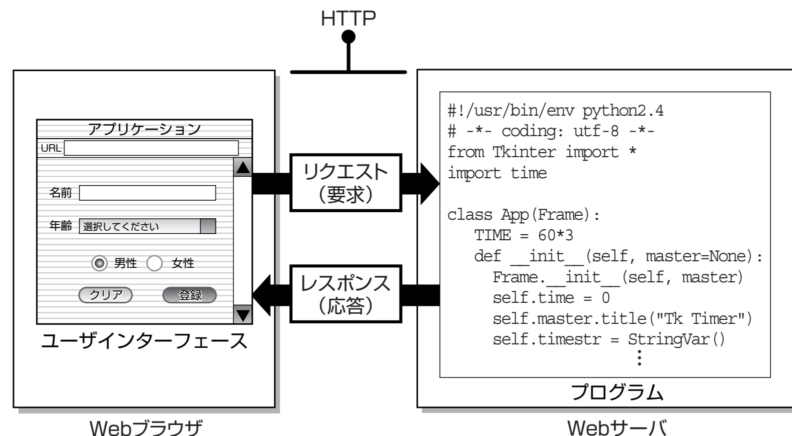
HTTPの詳細

これまで、Webアプリケーションが動作する際のクライアントとサーバの通信について、表面的に見える部分についてのみ解説してきました。しかし実際には、これまで見てきた以上に多くの情報がやりとりされています。ここでは、Webアプリケーションの動作について理解を深めるため、表面には見えない内部的な仕組みについて簡単に解説します。

Webを含め、たいていの通信ではルールや手順が決められています。決められた手順に従って通信を行うことで、いろいろな種類のサーバとクライアントで問題なく通信が行えるのです。そのようなルールのことを**プロトコル**と呼んでいます。

Webの通信に使われるプロトコルは**HTTP**（HyperText Transfer Protocol）と呼ばれています。前半のHT（HyperText）はHTMLの前半と同じで、他の文書へのリンクを埋め込める仕組みのことを指しています。後半のTP（Transfer Protocol）は通信のためのプロトコルというような意味です。

図01 Webの通信にはHTTPと呼ばれるプロトコルが使われる



Webアプリケーションはクライアント・サーバ型のアプリケーションです。通信には2種類があります。Webサーバからクライアントに向かう**レスポンス**が１つ。クライアント（Webブラウザ）からWebサーバに向かう**リクエスト**がもう１つの通信です。

05-01　レスポンスとして送られる文字列

Webサーバが返すレスポンスは主にHTMLや、画像などのデータです。HTMLは文字列データと呼ばれています。画像などはバイナリデータと呼ばれています。クライアント（Webブラウザ）はこのデータを受け取り、データを表示します。

Webアプリケーションを構成するプログラムでは、リクエストに応じてなんらかの処理を行い、結果をHTMLの文字列として組み立てます。プログラムは**標準出力**にHTMLを出力します。するとWebサーバが通信を行い、Webブラウザにレスポンスとして返します。

サーバから実際に送り出されるレスポンスには、HTMLのように表面に見えるデータ以外にも多くのデータが含まれています。以下に簡単な例を示します。

レスポンスの例：

```
HTTP/1.1 200 OK
Server: SimpleHTTP/0.6 Python/2.5
Date:  Sun、 1 Jul 2007 11:31:06 GMT
Content-type: text/html
Content-Length: 173

<html> ....
```

ステータスコード

1行目は**応答コード**と呼ばれている部分です。200というのは、処理が正常に行われたことを示す**ステータスコード**と呼ばれています。これ以外にも多くのステータスコードが決められています。

200番台のステータスコードは、レスポンスが正常に送り出されていることを示すステータスコードです。400番台はクライアント側に原因のあるエラーが発生したときに利用されます。たとえば、リクエストのURLにあるパスにファイルが見付からない場合は「404」という番号が返ります。Webサーバで動くプログラムでエラーが起こった場合などは、500番のステータスコードが返ります。500番台のステータスコードは、サーバ側が原因のエラーが起こったときに使われるステータスコードです。

レスポンスのヘッダ

2行目以降はヘッダフィールドと呼ばれる部分です。情報の種類を表す文字列**フィールド名**と、情報の内容を表す文字列**フィールド値**をコロン（:）で区切って並べています。ここにあるだけでなく、いろいろな種類のヘッダが状況に応じて追加されます。

空の行を１つ挟んで、Webブラウザに表示されるHTML文字列が続きます。この部分がWebブラウザに表示されるわけです。

ところで、ヘッダの中には見慣れた文字列が見えます。ヘッダの最後から2行目の部分です。この文字列は、Webサーバで動くPythonのプログラムに必ずありました。

```
print "Content-type: text/html¥n"
```

このように、print文でこのヘッダ行に相当する部分を表示していました。最後にあるのは改行文字です。print文は末尾に改行を追加しますので、ヘッダの後に１つ空の行ができることになります。プログラムに必ず書いていたおまじないは、実はレスポンスのヘッダ部分だったということになります。同様に、プログラムから他の種類のヘッダを出力することで、他の情報をWebブラウザに送り出すことができます。

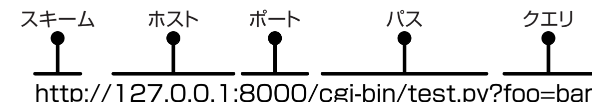
なお、**Content-Type**というヘッダは、レスポンスがどのような種類のデータであるかを指定しています。**text/html**という部分がデータの種類を指示していて、HTML文書であることを示しています。Webブラウザは、このヘッダを解釈して、レスポンスの扱いを決めるのです。image/jpegなどとすると、レスポンスのデータはJPEG画像として扱われます。

空の行を１つ表示することで、ヘッダが終わったことを表します。プログラムでヘッダを出力したあと、改行文字列（¥n）を出力していたのは、ヘッダの終わりを示すためです。

05-02　リクエストとして送られる文字列

WebブラウザにURLを入力すると、Webサーバに対してリクエストが送られます。URLは前半（スキームとホスト）と後半（パスとクエリ）に分かれます。前半をWebブラウザが知っているべき情報です。Webブラウザがスキームとホストを解釈し、リクエストを送信する方法と送信先がネットワークのどこにあるかを決めます。

図02　URLはいくつかの部分に分かれる



URLの文字列のうち、Webサーバにリクエストとして送られるのは、基本的に後半の**パス**と**クエリ**部分のみです。たとえば「http://127.0.0.1:8000/cgi-bin/foo.py?bar=baz」というリクエストが送られるとします。127.0.0.1は**ループバックアドレス**と呼ばれ自分自身を指すことはすでに解説しました。Webブラウザと同じマシンの8000番ポートに対して、以下のような文字列がリクエストとして送信されます。そう、リクエストも文字データなのです。

レスポンスの例：

```
GET /cgi-bin/foo.py?bar=baz HTTP/1.1
If-Modified-Since: Sun、 1 Jul 2007 11:31:06 GMT
User-Agent: ....
```

リクエストラインとヘッダ

リクエストのうち、1行目は**リクエストライン**と呼ばれています。リクエス

トラインには、リクエストの種類とパスやクエリなどの情報が記載されています。2行目以降はヘッダフィールドと呼ばれています。ヘッダフィールドには、主にブラウザが追加したいろいろな情報が記載されています。リクエストのヘッダフィールドと同じように、いろいろな付加情報が記載されています。

リクエストラインはURLとほぼ同じ内容です。それに対して、ヘッダフィールドは表面には見えません。リクエスト送信時に、Webブラウザがこっそりと埋め込んで送信する情報だからです。実際にWebブラウザから送信されるヘッダにはもっとたくさんの種類があります。

リクエストにあるヘッダの情報は、**環境変数**としてWebサーバ上で動くプログラムに受け渡されます。cgiモジュールには、この環境変数を整形して表示するための関数が用意されています。以下のようなプログラムをcgi-binフォルダに設置して、Webブラウザでアクセスしてみてください。環境変数として、たくさんの情報が表示されるはずです。

List01 environ.py

```
#!/usr/bin/env python

import cgi

print "Content-type: text/html¥n"
print cgi.print_environ()
```

図03 実行結果

環境変数の中には、Webサーバを実行している環境に設定された情報が含まれています。そのため、情報のすべてがリクエストに送られてくる情報ではありません。CONTENT_TYPEやHTTP_USER_AGENTなどの環境変数の多くは、リクエストのヘッダで送信された情報です。

HTTPメソッド

Webブラウザから送られるリクエストには何種類かあります。先ほどのリクエストでは、リクエストラインの先頭に**GET**という文字列が見えました。この部分は**メソッド**（method）と呼ばれていて、Webサーバに与えるリクエストの種類を示しています。Pythonで組み込み型やインスタンスオブジェクトに対して利用するメソッドと同じ言葉なので混乱するかもしれません。methodという英単語には「方法」「手法」という意味があります。ここでは、リクエストを送るための方法、という意味ととらえてください。なお、ここでいうメソッドは、<form>タグにアトリビュートとして埋め込んだmethodと同じことを指しています。

次に、クエリデータの送り方が異なる2つのメソッドについて解説したいと思います。

GETメソッド

Webサーバに対してレスポンスを要求するために、パスを指定したリクエストを発行するのに利用するのが**GETメソッド**です。HTMLファイルや画像ファイルのような静的ファイルをWebサーバに要求するときに主に利用します。

Webサーバ上で動くプログラムに値を渡すときには、パスの後ろに**クエリ**を追加します。クエリは、Pythonの辞書型と同じような構造をしていて、キーと値のペアをイコールでつなげる形で記述します。複数のペアがあるときにはアンド（&）で区切ります。

フォームを使ってリクエストを送信するとき、GETメソッドを利用したい場合は<form>タグのmethodアトリビュートに「GET」を指定します。<form>タグにmethodアトリビュートを指定しない場合はGETメソッドとなります。

クエリとURLエンコード

GETメソッドのクエリの中で利用できる文字列には制限があります。たとえば、アンド(&)やイコール(=)のような文字列は、クエリの中で**区切り文字**として利用されます。クエリの値としてこのような文字があると、クエリ中の区切りが正しく認識できなくなってしまいます。

このようなことを避けるため、クエリでは特定の文字列を**エスケープ**する、という決まりがあります。エスケープすべき文字列は**%+16進の文字コード**という形でエスケープします。

以下のような規則で、エスケープすべき文字とそうでない文字が決まっています。

● **エスケープしなくてよい文字列**

英字(大文字、小文字とも)、数字、ハイフン(-)、アンダースコア(_)やピリオド(.)などの一部記号

● **エスケープする文字**

アンド(&)やイコール(=)などURLで意味がある文字、空白文字や不等号(<>)などURLで使えない文字、日本語のようなマルチバイト文字列

フォームからGETメソッドでリクエストを送信する場合には、Webブラウザが自動的にURLエンコードを行います。また、一部のWebブラウザでは、URLの中にマルチバイトがあった場合に自動的にURLエンコードをするようです。

なお、プログラム側では、クエリの文字列を解析して**キー**と**値**に分割して処理をします。クエリの値はURLエンコードされているので、エンコードを解いて、元の文字列を取り出す処理をするわけです。

PythonでWebアプリケーションを作る場合、クエリの解析を行うことはほとんどありません。cgiモジュールの**FieldStorage()関数**を呼ぶときに、クエリの解析が自動的に行われるからです。クエリを解析した結果は、FieldStorage()関数の返すフィールドストレージオブジェクトに格納されます。

デコードとは逆に、クエリをエンコードするには、urllibモジュールの**urlencode()関数**を利用します。引数としてクエリを辞書にして渡すと、クエリをURLエンコードした文字列が返ってきます。以下に簡単なサンプルを示します。

urlencode()関数の使用例：

```
import urllib
querystr=urllib.urlencode({'foo':'bar& baz'})
print querystr
foo=bar%26+baz
```

POSTメソッド

POSTというのは**投稿をする**という意味の英単語です。このことからも分かると思いますが、Webサーバに対してデータを送信するときに利用するのがPOSTメソッドです。

GETメソッドを使う場合、URLにクエリを渡さなければなりません。URLの長さにはたいてい制限がありますし、あまり長いURLは好ましいとは言えません。大きなサイズのデータを送るにはGETメソッドは不向きです。大きなサイズのデータを送るときにはPOSTメソッドを利用します。

フォームを使ってPOSTメソッドでデータを送るには、methodアトリビュートに「POST」という文字列を指定します。POSTリクエストの場合、クエリのデータはリクエストのヘッダの後に改行を挟んで埋め込まれます。埋め込まれるデータは、クエリのキーと値のペアをURLエンコードした文字列です。

以下は簡単なPOSTリクエストの例です。「bar」というキーに対応する値として「baz」という文字列を指定しています。GETリクエストでURLに埋め込まれていたクエリ文字列が、ヘッダの直後に来ているのが分かります。

● POSTリクエスト時のデータ送信

```
POST /cgi-bin/foo.py HTTP/1.1
If-Modified-Since: SunßA 1 Jul 2007 11:31:06 GMT
User-Agent: ....

bar=baz
```

cgiモジュールのFiedlStorage()関数では、POSTメソッドのリクエストを解析して、クエリをオブジェクトに変換します。変換は自動的に行われるので、プログラム側でPOSTメソッドが使われているかGETメソッドが使われてい

るかを意識する必要はあまりありません。

GETメソッド、POSTメソッドの利点、欠点

前述のとおり、クエリを送るメソッドにはGETとPOST2つの種類があります。GETメソッドは、特定の性質を持ったデータを取得したいときに利用されます。POSTメソッドは、データの新規作成や更新のために利用されます。

GETとPOSTにはそれぞれ利点と欠点があります。2つのメソッドの最大の違いは、クエリがURLの一部として残るかどうか、ということです。どのような目的でリクエストを送るかによって、2つのメソッドを使い分けることになります。

GETメソッドではクエリの内容がURLに記載されます。このため、クエリの内容をURLの文字列として保存できます。

私たちが最もよく目にするGETメソッドのリクエストは、検索エンジンの検索結果です。以下はGoogleで「Python」という言葉を検索したときのURLです（一部のクエリを省略しています）。

```
http://www.google.co.jp/search?hl=ja&q=python&ie=UTF-8&oe=UTF-8
```

クエリとして検索語そのものが埋め込まれているのが分かります。リクエストに必要な情報がすべてURLに埋め込まれているので、この文字列をリンクとしてHTMLに埋め込むことができます。

検索結果がたくさんある場合に「次の結果を表示」というようなリンクが表示されます。このようなリンクを**ページネーション**と呼びます。検索エンジンのページネーションでは、クエリがリンクに埋め込まれています。GETメソッドでクエリを発行して、同じ検索語の別の結果を表示するリンクを作っているわけです。また、この文字列をそのままメールに貼り付けることもできます。

一方、POSTメソッドではクエリがリクエストの中に見えないように埋め込まれてしまいます。このため、POSTリクエスト後のURLをページネーションのリンクとしてそのまま利用することはできませんし、メールに貼り付けても正しい結果が得られません。

GETリクエストにも欠点があります。URLには長さの制限があるため、大きなサイズのクエリが送信できないのです。URLのサイズ制限はWebサーバによって異なりますが、1キロバイト（1024）からせいぜい数キロバイトほどです。このため、数十キロバイトから数メガバイトもあるような大きなサイズのデータをGETメソッドで送信することはできません。

大きなサイズのデータを送りたいときはPOSTメソッドを使うことになります。POSTメソッドで送るデータのサイズには事実上制限がありません。POSTメソッドを使うと、長い文字列や文章だけでなく、画像、動画データをWebサーバやWebアプリケーションに送ることができます。

また、POSTメソッドは外部に見せたくないデータを隠す意味でも使われます。POSTメソッドではクエリのデータが表面上には見えないようにWebサーバに送られます。パスワードのような文字列を、URLのように目に触れる場所に出さずにWebサーバに送信する、という目的で利用されることがあります。

CHAPTER 06

RSSリーダーを作る

これまで、Pythonを使ってWebアプリケーションを作るために必要となる、基本的な知識について解説してきました。ここでは、これまで学んだ知識を使って、より本格的なWebアプリケーションを作ってみましょう。

ここで作るのは簡易なRSSリーダーです。RSSのURLを入力すると、内容を読み込み整形して表示します。

実際にWebアプリケーションを作る前に、開発に利用するクラスや関数などを作成します。他のWebアプリケーションを作る際にも使い回しできるように配慮したクラスや関数を作ります。

06-01　リクエストの処理

まず最初に、リクエストをうまく扱う方法について考えてみましょう。

リクエストはWebアプリケーションに対する入力です。Webブラウザ上に表示されたフォームなどのUIから、文字列のデータとしてWebサーバに送られてきます。Webアプリケーション側では、主にクエリとして情報を受け取ります。実際には、クエリ以外にも多くの情報がWebブラウザからサーバに送られていますが、今の段階ではクエリ以外の情報を使うことはありません。

Webアプリケーションでは、UIから送られたクエリデータを解析、変換する作業を毎回行います。PythonでWebアプリケーションを書くときには、cgiモジュールのFieldStorage()関数を使います。FieldStorage()関数では、リクエストの内容をPythonの文字列などに変換します。変換されたデータを使って、処理を行うわけです。

プログラムで毎回行う**定型処理**は、モジュールやクラスのような形でまとめておくと便利です。

「値を与え、結果を返す」という単純な処理であれば関数として実装するべきです。しかし今回は、クエリのデータをどこかに保存する必要があります。そのような用途に利用するためには、クラスを定義しておくと便利です。

Pythonのクラスでは、アトリビュートにいろいろななデータを保存しておくことができます。クエリのデータをアトリビュートに保存しておけば、いろいろな場所で利用することができます。また、リクエストのデータを処理するための定型処理をメソッドとして定義しておく、といった使い方もできます。

では、リクエストのデータを取り扱うためのクラスを作ってみましょう。Pythonの場合、クラス名は大文字から始めるという規則になっていますので**Request**という名前のクラスを作ってみることにします。

Requestクラスを実装する

Pythonのクラスでは、__init__()という初期化メソッドが利用できます。クラスインスタンスを作るときに呼ばれるメソッドです。このメソッドの中で、クラスインスタンスの初期化を行います。Requestクラスではリクエストに送られてくるクエリを取り出し、Pythonのデータ型に変換してインスタンスに保存しておきます。初期化メソッドではこの処理を行うことになります。

以下が、現時点でのRequestクラスの定義になります。初期化メソッドがあるだけの、とてもシンプルなクラスです。

List01 Requestクラスの定義(httphandler.py)

```
# 標準モジュールをimportする
import cgi
import os

class Request(object):
    """
    HTTPのリクエストをハンドリングするクラス
    CGI側でインスタンスを生成することによって利用する
    クエリデータや環境変数へのアクセス,主要ヘッダへの
    アクセス用メソッドを提供
```

```
    """

    def __init__(self, environ=os.environ):
        """
        インスタンスの初期化メソッド
        クエリ,環境変数をアトリビュートとして保持する
        """
        self.form=cgi.FieldStorage()
        self.environ=environ
```

Pythonのメソッド定義では、第1引数に必ず**self**という引数を指定します。引数selfには、クラスから作られるインスタンスオブジェクトが渡ってきます。メソッドの内部でインスタンスのアトリビュートを利用したいときには、selfからドットを挟んでアトリビュート名を指定します。

新しいアトリビュートを作るときには**代入**を行います。変数を定義するときと同じです。Pythonには、JavaやC++のようなオブジェクト指向言語にある「メンバ変数」のような仕組みがありません。そのため、クラスインスタンスに必須なアトリビュートは、初期化メソッドで定義しておくようにします。

初期化メソッドでは、第1引数self以外にenvironという引数が指定されています。この引数には、デフォルト値として環境変数を渡しています。環境変数には、Webアプリケーションが受け取ることができるさまざまな情報がつまっています。あとあと利用することがあるはずですので、環境変数の情報もインスタンスのアトリビュートして保存しています。

クラスの定義、メソッドの定義には、三重引用符で囲まれた文字列が見えます。この文字列は**ドキュメンテーション文字列**と呼ばれています。ドキュメンテーション文字列には、クラスや関数に関する解説や使い方を書いておくようにします。

Requestクラスの使い方

Webアプリケーションでは、「req=Request()」のようにしてRequestクラスのインスタンスオブジェクトを作るようにします。クラスを作ることによって、リクエストとして送られてくる情報がインスタンスのアトリビュートに入ります。プログラムでは、インスタンスオブジェクトを経由してリクエストの情報にアクセスし、処理を実行するわけです。

06-02　レスポンスの処理

Webアプリケーションが入力を受け取り、処理をした結果としてレスポンスを返します。Webアプリケーションの場合、レスポンスもほとんどの場合HTMLのような文字列となります。レスポンスとして返された文字列をWebブラウザのようなクライアントが受け取り、結果を表示します。

Webアプリケーションが返すレスポンスには、HTMLのような文字列の他にも多くの情報を埋め込めます。**ヘッダ**と呼ばれている部分に、レスポンスを返した日付などの情報を埋め込むことができるのです。

これまでのサンプルでは、ヘッダを含んだ文字列を、Pythonのプログラムから出力していました。最初に必ず、**Contet-type**から始まる行がありました。この部分がヘッダになっています。ヘッダの後、HTMLを送信することでレスポンスを出力しています。毎回、似たような内容の文字列を出力するわけです。この部分を含め、レスポンスをスマートに扱う方法について考えてみましょう。

レスポンスとして送るべきヘッダには、レスポンスの種類を示すContent-typeの他にもいくつかの種類があります。ヘッダの内容によって、レスポンスを受け取るクライアントの動きをコントロールすることもあります。ですので、必要に応じてヘッダを付け加えたり、ヘッダの内容を変更できるようになっているのが理想です。

まとめると、レスポンスには、ヘッダの内容、およびレスポンス本文となる文字列を保存する必要がある、ということになります。データを保存する必要があるので、レスポンスもクラスとして実装することにしましょう。クラス名は「Response」です。

Responseクラスを実装する

レスポンスとして返す文字列をスマートに扱うために、Responseクラスを実装してみましょう。レスポンスクラスのインスタンスには、ヘッダと、Webブラウザに表示するHTML文字列を保存できるようにします。

ヘッダは、**名前**と**値**という2種類の文字列のペアで表現します。このようなデータを扱うためには、辞書オブジェクトを使えばいいでしょう。HTML文字列は、文字列として保存しておけばよいでしょう。2つのデータはイン

スタンスオブジェクトのアトリビュートとして保存します。初期化メソッドでアトリビュートを初期化しておくわけです。

ヘッダには複数の種類があります。レスポンスとして送信すべきヘッダのうち何種類かは、デフォルトの状態を決めることができたり、プログラム側で作ることができます。そのようなヘッダは、クラスのメソッドで自動的に作るようにします。

プログラム側では、あらかじめResponseクラスのインスタンスオブジェクトを作っておきます。インスタンスに対して、ヘッダを追加したりレスポンス本文を登録したり、という処理をするわけです。最終的に、ヘッダと本文を含んだレスポンス文字列全体を得るためには、クラスのメソッドを利用します。

以下が、現時点でのResponseクラスの初期化メソッドです。ヘッダのうち、レスポンスの種類を指定するヘッダをあらかじめ定義しておきます。また、レスポンス本文となるアトリビュートを空の文字列として初期化しています。レスポンスの1行目として返すステータス行のために、ステータスコードとステータスメッセージをアトリビュートとして定義しています。

List02 Responseクラスの初期化メソッド(httphandler.py)

```
class Response(object):
    """
    HTTPのレスポンスをハンドリングするクラス
    レスポンスを送る前にインスタンスを生成して利用する
    レスポンスやヘッダの内容の保持,ヘッダを含めたレスポンスの
    送信を行う
    """

    def __init__(self, charset='utf-8'):
        """
        インスタンスの初期化メソッド
        ヘッダ用の辞書,本文用の文字列などを初期化する
        """
        self.headers={'Content-type':'text/html;charset=%s' %
                    charset}
        self.body=""
        self.status=200
        self.status_message=''
                    :
```

ヘッダの追加用メソッド

次に、レスポンス・ヘッダを追加したり、取り出したりするメソッドを定義します。Webアプリケーション側では、必要に応じてこのメソッドを使い、ヘッダを登録します。

List03 ヘッダの制御用メソッド(httphandler.py)

```
                    :
    def set_header(self, name, value):
        """
        レスポンスのヘッダを設定する
        """
        self.headers[name]=value

    def get_header(self, name):
        """
        設定済みのレスポンス用ヘッダを返す
        """
        return self.headers.get(name, None)
                    :
```

メソッドの内部では、アトリビュートの辞書を使って処理をしています。ヘッダを取得するget_header()メソッドでは、ヘッダが定義されていない場合は、Noneを返すようにしています。辞書の未定義のキーに対してself.header[name]のようにアクセスしようとすると、KeyErrorという例外が発生します。get()メソッドを使うことで、例外が発生しないように処理をしています。

次に、レスポンス本文を設定するメソッドと、レスポンス文字列全体を作るメソッドを見てみましょう。

List04 本文の登録とレスポンス文字列の生成(httphandler.py)

```
                    :
    def set_body(self, bodystr):
        """
        レスポンスとして出力する本文の文字列を返す
        """
        self.body=bodystr

    def make_output(self, timestamp=None):
```

```
        """
        ヘッダと本文を含めたレスポンス文字列を作る
        """
        if timestamp is None:
            timestamp = time.time()
        year, month, day, hh, mm, ss, wd, y, z = time.gmtime(
        timestamp)
        dtstr="%s, %02d %3s %4d %02d:%02d:%02d GMT" % (
        _weekdayname[wd], day, _monthname[month], year, hh, mm, ss)
        self.set_header("Last-Modified", dtstr)
        headers='¥n'.join(["%s: %s" % (k, v)
                        for k,v in self.headers.items()])
        return headers+'¥n¥n'+self.body

    def __str__(self):
        """
        リクエストを文字列に変換する
        """
        return self.make_output().encode('utf-8')
                    ⋮
```

レスポンス文字列を設定するset_body()はとても簡単なメソッドです。引数として受け取ったオブジェクトを、インスタンスオブジェクトのアトリビュートに代入するだけのメソッドです。

make_output()は、ヘッダと本文を含めたレスポンス文字列全体を作るメソッドです。初期化メソッドではContent-Typeヘッダが追加されています。また、set_header()などを通じて追加されたヘッダの他に、別のヘッダを追加しています。

レスポンスを返す時間を表す**Last-Modified**ヘッダは、決められたフォーマットに沿って文字列を作らなければなりません。ここでは、timeモジュールを使って現在日時を文字列に変換しています。

最後に、ヘッダ全体を文字列に変換し、改行を挟んでレスポンス本文を連結して１つの文字列を作っています。

最後に定義されている__str__()メソッドは**特殊メソッド**と呼ばれるメソッドです。インスタンスオブジェクトを文字列に変換しようとするときに呼ばれるメソッドです。Responseクラスでは、レスポンス文字列全体を返します。このメソッドを定義しておくと、インスタンスオブジェクトを文字列に変換するだけで、レスポンス文字列を得ることができるわけです。

そのほかの処理

ここで定義したRequestクラスとResponseクラスを、「httphandler.py」という１つのファイルにまとめます。Pythonでは、スクリプトファイルがモジュールとなります。Webアプリケーションでは、このモジュールからクラスをインポートして、利用するわけです。

またhttphanderモジュールには、レスポンス本文としてよく使う<html>〜</html>のような文字列を返す関数を定義しておきます。HTMLの**外枠**となる定型文字列を返す関数を定義して、Webアプリケーションて利用するためです。この関数はget_htmltemplate()とします。

List05 get_htmltemplate()関数（httphandler.py）

```
def get_htmltemplate():
    """
    レスポンスとして返すHTMLのうち，定型部分を返す
    """
    html_body = u"""
    <html>
      <head>
        <meta http-equiv="content-type"
              content="text/html;charset=utf-8" />
      </head>
      <body>
      %s
      </body>
    </html>"""
    return html_body
```

そのほか、Last-Modifiedヘッダを作成するためにtimeモジュールの読み込みと、以下のような配列変数の定義も必要です。

List06 timeモジュールと配列変数の定義（httphandler.py）

```
import time

_weekdayname = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
_monthname = [None,
              "Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
```

6-3 簡易RSSリーダーを作る

さて、いよいよリクエストとレスポンスを取り扱うための2つのクラスを活用し、RSSリーダーを作ってみましょう。

目標とするのは、フォームにRSSのURLを入力すると、RSSを読み込んで整形、タイトルやリンクなどを表示する簡易なものです。フォームの表示、RSSの整形表示も１つのスクリプトで行うようにします。また、読み込む対象とするのは**RSS 2.0**のみとします。RSSにはさまざまな規格があり、方言も多いため、手軽に扱えて方言の少ないRSSのみを扱うものとします。また、エンコードに関する処理を最小限度に抑えるため、RSSのエンコードはASCIIとUTF-8のみに限定します。

RSSを読み込む

RSSを読み込み、Pythonで処理をするためには2つの処理が必要です。まず、Webの通信を通してRSSを文字列として取り込む必要があります。その後、読み込んだRSSを要素に分解してPythonのオブジェクトに変換します。

RSSをWebから読み込むためには、標準モジュールのurllibモジュールを使えばよいでしょう。読み込んだRSSは、XMLと呼ばれる構造を持った文字列となります。XMLを取り扱うためには、**ElementTree**というモジュールを使うと便利です。ElementTreeはPython 2.5から標準モジュールとしてPythonに組み込まれていますので、別途インストールする必要もありません。

RSSを読み込み、辞書のリストとして要素を取り出すための関数を定義します。この関数は別のWebアプリケーションで利用することがあるかもしれません。関数を「rssparser.py」という名前のスクリプトファイルに書き出し、外部のプログラムからモジュールを通して利用できるようにします。

以下がモジュールの定義です。

List07 rssparser.py

```
#!/usr/bin/env python
# coding: utf-8

from xml.etree.ElementTree import ElementTree
from urllib import urlopen

def parse_rss(url):
    """
    RSS 2.0をパースして、辞書のリストを返す
    """
    rss=ElementTree(file=urlopen(url))
    root=rss.getroot()
    rsslist=[]
    # RSS 2.0のitemエレメントだけを抜き出す
    for item in [ x for x in root.getiterator()
                        if "item" in x.tag]:
        rssdict={}
        for elem in item.getiterator():
            for k in ['link', 'title', 'description', 'author',
                                             'pubDate']:
                if k in elem.tag:
                    rssdict[k]=elem.text
                else:
                    rssdict[k]=rssdict.get(k, "N/A")
        rsslist.append(rssdict)
    return rsslist
```

RSS 2.0では、ブログ記事などのタイトル、リンクなどの要素が**item**というエレメントの中に収まっています。このスクリプトでは、RSS 2.0相当のXML文字列を先頭から読み込み、itemエレメントを探し出す、という処理をしています。itemエレメントを探し出したら、リンクやタイトルといった要素を辞書に登録します。

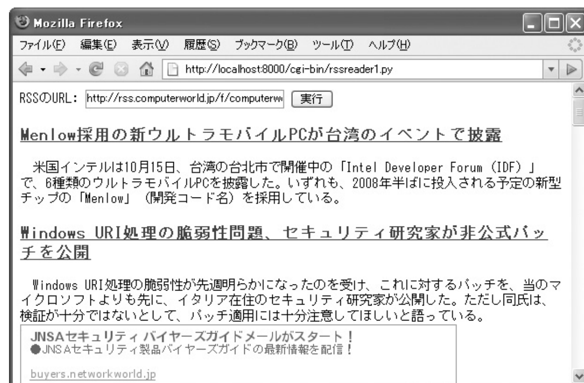
１つの記事に対する要素が１つの辞書になります。複数itemエレメントがある場合は、複数の辞書がリストに登録されます。関数の戻り値となるのは、itemエレメントの内容を収めた辞書のリストです。

Webアプリケーションを作る

リクエスト、レスポンスをスマートに扱うためのクラスを作りました。また、RSS 2.0を読み込み、Pythonのオブジェクトに変換するモジュールを作りました。いままで作ってきたものを組み合わせて、Webアプリケーションの本体となるスクリプトを作りましょう。ファイル名は「rssreader1.py」とします。httphandler.py、rssparser.pyとともに、cgi-binフォルダに設置しま

す。PythonのWebサーバを立ち上げて、Webブラウザで/cgi-bin/rssreader1.pyというURLにアクセスすることでWebアプリケーションを利用できます。

図01 RSSリーダーの実行画面



以下がWebアプリケーション用のプログラムです。これまでのサンプルプログラムは文字列の固まりというたたずまいでした。このプログラムではリクエストやレスポンスの扱いを工夫していることもあり、プログラムらしく見えると思います。

List08 rssreader1.py

```
#!/usr/bin/env python
# coding: utf-8

from rssparser import parse_rss
from httphandler import Request, Response, get_htmltemplate
import cgitb; cgitb.enable()                                    ❶

form_body=u"""
  <form method="POST" action="/cgi-bin/rssreader1.py">
    RSSのURL:
    <input type="text" size="40" name="url" value="%s"/>
    <input type="submit" />
  </form>"""

rss_parts=u"""
<h3><a href="%(link)s">%(title)s</a></h3>                       ❸
<p>%(description)s</p>
"""

content=u"URLを入力してください"
req=Request()                                                   ❷
if req.form.has_key('url'):
    try:
        rss_list=parse_rss(req.form['url'].value)
        content=""
        for d in rss_list:
            content+=rss_parts%d
    except:
        pass

res=Response()                                                  ❹
body=form_body%req.form.getvalue('url', '')
body+=content
res.set_body(get_htmltemplate()%body)
print res                                                       ❺
```

プログラムの冒頭では、処理に利用するモジュールなどをインポートしています。インポートブロックの最後の行で**cgitb**というモジュールをインポートしています。このモジュールは、Webアプリケーションでエラーが起こったとき、分かりやすいエラーを表示するためのモジュールです。モジュールをインポートし、**cgitb.enable()**という関数を呼ぶと、エラーが起こったときにエラーの位置などを色つきで表示します（❶）。

モジュール定義の後は、レスポンスとして出力する文字列を定義しています。form_bodyという変数にはRSSのURLを入力するフォームを表示するためのHTMLを定義しています。rss_partsには、RSSを整形表示するための文字列を定義しています。表示すべき要素が複数ある場合には、この文字列に必要な要素を埋め込み、繰り返し利用します。

後半以降はRSSを整形表示するための処理を行っています。まず、フォームに入力されたRSSのURLを取得するために、Requestクラスのインスタンスオブジェクトを作っています（❷）。

「url」というキーがあった場合には、先ほど作った関数を使ってRSSを

Pythonのオブジェクトに変換します。変換したオブジェクトを元に、表示用のHTML文字列を作って足していきます。

rss_partsという変数に入った文字列は、RSSの要素を埋め込む**テンプレート**のような役割をしています。文字列の埋め込みには**フォーマット文字列**機能を使っています。テンプレートには「% (link) s」のように辞書のキーが埋め込まれています（❸）。テンプレートと辞書を組み合わせると、キーを元に置換を行います。

表示するHTMLができ上がったら、Responseクラスのインスタンスオブジェクトを作ります（❹）。メソッドを使ってレスポンス本文をインスタンスに設定します。

最後に、Responseクラスのインスタンスを文字列として扱いprint文で表示します（❺）。こうすると、Responseクラスに定義された__str__()メソッドが呼び出され、ヘッダを含めたレスポンス文字列全体がレスポンスとして送り出されます。これまでのサンプルでは、ヘッダを含めてprint文で表示していました。それに比べると、プログラムの書き方がずっとスマートになっているのが分かると思います。

Webアプリケーション開発とヒアドキュメント

これまで見てきたように、Webアプリケーションでは多くの文字列処理を行います。HTMLやレスポンスとして送信するヘッダのような文字列を扱うプログラムの内部に書くときには注意が必要です。プログラム自体も文字列なので、プログラムの内部にさまざまな種類の文字列が散らばってしまうと、プログラムが見づらくなってしまいます。また、プログラムのいろいろな場所に文字列が散らばっているのでは、プログラムの修正が難しくなってしまいます。

事実、このような問題はWebアプリケーションが登場した初期にはよく起こっていました。HTMLのような文字列とプログラムが混在したプログラムが多く書かれていました。プログラムを修正したり拡張するとき、開発者はまずどの部分がプログラムで、どの部分がレスポンスに使う文字列かを探り出す必要がありました。機能追加や修正が容易に行えないのでは、大規模なWebアプリケーションを作ることは難しくなってきます。このような手法は**ヒアドキュメント**などと呼ばれ、Webアプリケーションの開発では禁じ手とされています。

読みやすく、拡張や修正がしやすいWebアプリケーションを書くためには、文字列をプログラムに埋め込む方法に工夫が必要です。たとえば、繰り返し使う文字列は1箇所にまとめる。文字列の定義とプログラムを分ける。このような工夫をすることで、Webアプリケーションのプログラムはずっと見やすく、修正がしやすくなります。

このような工夫をすることが、効率的にWebアプリケーションを開発するための第一歩と言えます。最近では、より進んだ考え方や手法を活用して、もっと効率的にWebアプリケーションを開発する手法が利用されています。どのような手法があるのかについては、のちほど解説します。

CHAPTER 07
Webアプリケーションとデータの保存

本書の最初では、Webアプリケーションの基本的な動作原理について解説をしました。クライアントとサーバの間で、リクエストとレスポンスを繰り返して処理を続けていくというのが、Webアプリケーションの基本的な動作の流れです。

実際にリクエストやレスポンスとしてやりとりされるデータは、形式を持ったテキストデータであるということについても学びました。データの受け渡しは、HTTPという通信手順（プロトコル）に従って行われます。表面的に見えるデータ以外にも、たくさんの情報がヘッダという形でやりとりされています。

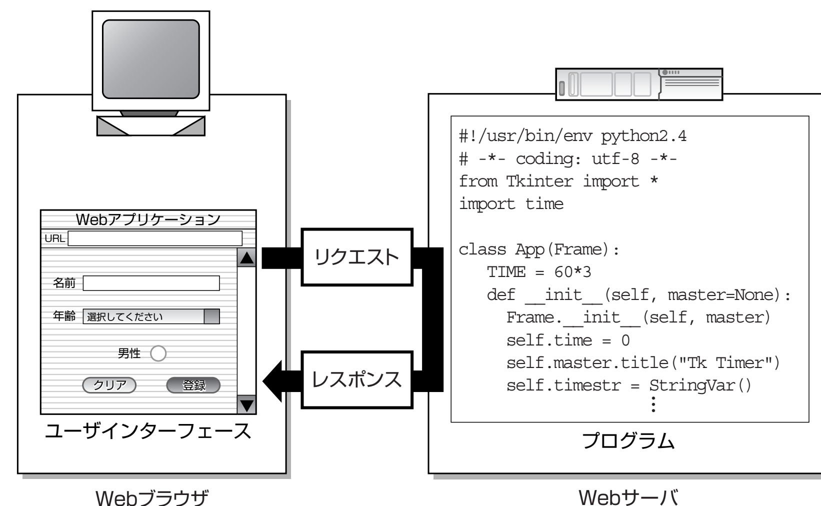
07-01　Webアプリケーションとセッション

Webブラウザがリクエストを送り、リクエストを受け取ったWebサーバがレスポンスを返す。Webの処理の基本となるこの一連の流れを**セッション**と呼びます。Webアプリケーションの処理は、このセッションを１つの単位として進んでいくことになります。

Webアプリケーションに指示を与えるためには**リクエスト**を使います。たとえば、Webブラウザ上に表示したフォームに文字列などを入力して、リクエストを**POST**します。すると、フォームに入力したデータがリクエストに乗ってWebサーバに送られます。Webサーバ上のプログラムで、リクエストに乗ったデータを解析して、処理を行います。処理の結果は、レスポンスとしてWebブラウザに戻っていきます。

Webブラウザから送られたデータは、ネットワーク通信の上に乗って送られています。通信はたいてい一瞬で終わり、Webサーバがリクエストを受け取った後には消えてしまいます。同じ内容のデータを送信するためには、フォームに同じ内容を再度記入するなどして、同じ内容のデータを送信する必要があります。POSTリクエストでなく、GETリクエストを使えば、URLにリクエストの内容が残ります。何度も同じ内容を送信するのならGETリクエストを使えばよいわけですが、リクエストを再度送信する必要がある、という点では同じことです。

図01　リクエストとして送られた情報はすぐに消えてしまう



また、Webアプリケーションでは、多くの場合リクエストを受け取ったときにプログラムが動き出します。レスポンスを送信した後は、プログラムは終了します。プログラム自体が終了しますので、プログラムの内部で保存していた変数やオブジェクトは消えてしまいます。後で利用したいデータは、どこかに保存しておかなければならないのです。

このように、リクエストとして送ったデータや、Webアプリケーション内部で作成したデータは、特別な処理をしないと消えてしまいます。Webのデータ通信の仕組みには、基本的にデータを保存するための方法がありません。一般的なWebアプリケーションでは、過去にリクエストとして受け取った情報を再利用したい場面がよくあります。過去に投稿したブログの記事をフォ

ームで再編集する、またはWebアプリケーションの個人用設定を保存しておく。このような処理を実現するためには、Webサーバの側でデータを保存する必要があるのです。

標準ライブラリを使ってデータを保存する - pickleを使う

PythonのWebアプリケーションで、リクエストとして受け取ったデータを保存するにはどうすればよいでしょうか。プログラムで受け取ったデータを消えないように残し、後で再利用できるようにするには、ファイルにデータを保存すれば良さそうです。リクエストとして受け取ったデータをファイルに書き出し、必要に応じて読み込んで再現する、という処理をするわけです。

データの保存とシリアライズ

プログラムのデータをファイルに書き出すとき、必ず考えなければならない問題があります。ファイルというのは一種の**巻きもの**のようなものです。必ず前から後ろに向かって読み進めていきます。

文字を順番に並べた文字列のようなデータであれば、ファイルにそのまま保存をして、元の状態を再現することができます。しかし、Pythonのリストや辞書など、構造を持ったデータの場合はそうはいきません。構造を持ったデータをファイルに書き出すためには、一度前から後ろに読めるように変換を行う必要があります。このような処理のことを**シリアライズ**と呼んでいます。

たとえば、リストをファイルに書き出すことを考えましょう。ファイルに書き込むのは文字列のような一次元のデータです。リストをいったん文字列に変換し、必要に応じて復元する方法を考えます。次のような処理をするとします。

- **リストをファイルに書き出す**

リストの要素を、カンマ(,)のような区切り文字を使ってファイルに書き出す。

- **ファイルからリストを取り出す**

ファイルから文字列を読み込み、カンマで区切った文字列を分割する。

たとえば、文字列だけで構成されたリストであれば、この方法でリストのシリアライズと復元が行えます。ただし、リストの文字列にカンマが含まれていると、区切りがおかしくなってしまいます。また、文字列以外のデータ、たとえば数値や辞書などを要素として持つ場合は正しく処理が行われないはずです。

文字列だけでなく、数値がリストの要素にあったらどうなるでしょうか。または辞書のような複雑な構造を持つデータを要素に持つリストを完全にシリアライズするためには、複雑なプログラムを書かなければなりません。いずれにしても、簡易な方法では扱えるデータの種類に制限ができてしまいます。

標準モジュールpickleを使ってシリアライズを行う

データをファイルに保存するこためには、前段階としてシリアライズをする必要があります。いろいろな種類のデータを、過不足なくシリアライズする処理を作るのは意外と大変なのです。

Pythonの標準モジュールには、組み込み型を含むいろいろなデータをシリアライズする**pickle**というモジュールが備わっています。ここでは、piclkeを使ってPythonのデータをファイルに保存する方法を検討してみましょう。

pickleを使うと、Pythonのいろいろなデータを文字列に変換することができます。また、変換した文字列を元に、元のデータを復元することができます。Webアプリケーションで保存したいデータを一度文字列にすることで、ファイルに保存しやすくなります。また、ファイルに保存した文字列をpickleモジュールの関数を使って処理することで、元のオブジェクトを復元することができます。

pickleモジュールには、大まかに分けて2種類の関数があります。文字列をベースとして、Pythonのオブジェクトをシリアライズ、復元する関数と、ファイルをベースに処理を行う関数です。

- **dumps(obj)、dump(obj、 file)**

Pythonのオブジェクトを引数に渡し、シリアライズを行う関数です。dumps()は、シリアライズした結果を文字列として返します。dump()は、ファイルオブジェクトを引数に添え、シリアライズした結果をファイルに書

き出します。

● **loads (string)、load (file)**

dumps()、dump()でシリアライズした文字列から、Pythonのオブジェクトを復元する関数です。loads()は、Pythonのオブジェクトをシリアライズした文字列を引数として渡します。load()はシリアライズした結果を書き出したファイルを引数として渡し、Pythonのオブジェクトを復元します。

インタラクティブシェルを使って、実際にシリアライズと復元の過程を試してみましょう。まず、数値と文字列を要素に含むリストを定義します。その後、dumps()関数を使ってオブジェクトをシリアライズした文字列を取り出しています。最後に、オブジェクトをシリアライズした文字列から元のリストを復元しています。pickleを使うと、リストだけでなく、辞書やリストのリスト、クラスといった複雑なオブジェクトをシリアライズできます。

● シリアライズと復元

```
>>> import pickle
>>> l=[1, 2, "three", "four"]
>>> ps=pickle.dumps(l)
>>> print ps
(lp0
I1
aI2
aS'three'
p1
aS'four'
p2
a.
>>> pl=pickle.loads(ps)
>>> print pl
[1, 2, 'three', 'four']
```

Webアプリケーションでデータを保存する

では実際に、データを保存するWebアプリケーションを書いてみましょう。好きな軽量言語について投票し、投票結果を表示するWebアプリケーションを書いてみます。RSSリーダーと一緒に作った、リクエストとレスポンスをスマートに扱うためのクラスを活用してみましょう。

図02 好きな言語の投票を行い、結果を保存するWebアプリケーション

簡略化のため、投票用のフォームと、投票結果は1つのプログラムで表示するようにします。投票用のUIとしてはラジオボタンを使います。投票結果は、CSSを活用して棒グラフとして表示します。プログラムでは、投票用のフォームと投票結果の棒グラフを表示するHTMLを動的に生成し、表示するわけです。

プログラムの内部では、投票をした結果は、言語の名前をキーにしたPythonの辞書オブジェクトとして扱います。フォームで投票を受けると、選択された言語の名前に相当するキーの値として保存されている数値に1を足します。その後、辞書オブジェクトをpickleモジュールを使いシリアライズして文字列に変換し、ファイルに保存します。保存したファイルに書かれた文字列を、pickleモジュールを使って再度辞書オブジェクトに変換すれば、直前に投票した内容を再利用できます。

以下のプログラムが、投票を行い、結果をファイルに保存するWebアプリケーションです。先ほど作成したhttpdhandler.pyというモジュールをおなじディレクトリに置いた状態で使います。

List01 picklepole.py

```
#!/usr/bin/env python
# coding: utf-8

import pickle
from httphandler import Request, Response, get_htmltemplate
```

```
import cgitb; cgitb.enable()

form_body=u"""
  <form method="POST" action="/cgi-bin/picklepole.py">
    好きな軽量言語は?<br />
    %s
    <input type="submit" />
  </form>"""

radio_parts=u"""
<input type="radio" name="language" value="%s" />%s
<div style="border-left: solid %sem red; ">%s</div>
"""

lang_dic={}                                              ❶
try:
    f=open('./favorite_langage.dat')
    lang_dic=pickle.load(f)
except IOError:
    pass

content=""
req=Request()
if req.form.has_key('language'):
    lang=req.form['language'].value
    lang_dic[lang]=lang_dic.get(lang, 0)+1

f=open('./favorite_langage.dat', 'w')                    ❷
pickle.dump(lang_dic, f)

for lang in ['Perl', 'PHP', 'Python', 'Ruby']:           ❸
    num=lang_dic.get(lang, 0)
    content+=radio_parts%(lang, lang, num, num)

res=Response()
body=form_body%content
res.set_body(get_htmltemplate()%body)
print res
```

プログラムの前半はモジュールのインポート、および表示に利用するHTMLのテンプレート文字列を定義している部分です。その後、投票結果を保存するため、lang_dicという変数に辞書オブジェクトを代入しています(❶)。もし、前回投票時に保存したファイルがあれば、pickleモジュールを使って前回の投票内容を復元します。

その後、Requestクラスのインスタンスオブジェクトを生成して、クエリから投票内容を取り出します。投票の内容によって、結果を保存するための辞書を更新します。辞書を更新したら、更新した辞書をpickleモジュールを使いシリアライズ、あとで利用できるように、文字列としてファイルに書き出します(❷)。

最後はWebブラウザに表示するUIと結果の棒グラフをHTMLの文字列として組み立てています(❸)。HTML文字列ができ上がったら、Responseクラスのインスタンスを生成して、レスポンスを作って返します。

これまでのサンプルでは、リクエストとして受けた情報をその場で処理してレスポンスとなるHTML文字列を組み立てていました。このプログラムでは、受け取った情報をシリアライズし、毎回ファイルに保存しています。Webアプリケーションでは、このようにデータの保存を行って初めて、データを再利用できるのです。

pickle利用とマルチスレッド

pickleを使うと、Pythonのオブジェクトそのものを文字列に変換し、時間をおいて元の状態を復元できます。アプリケーションでデータを保存したいときに利用すると便利なモジュールです。

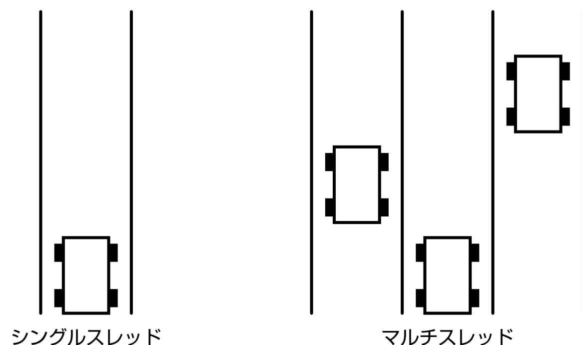
ただし、Webアプリケーションでpickleモジュールを使いファイルにオブジェクトの内容を保存するときには、十分に注意する必要があります。

今使っているPythonのWebサーバでは、同時に１つのリクエストしか受け付けないようになっています。しかし、たいていのWebアプリケーションでは、同じプログラムが複数同時に動くようになっています。複数のリクエストを同時に受け付けることができるようになっているわけです。

Webアプリケーションが同時に動く様子を、道路の車線に例えてみると分かりやすいと思います。Webブラウザから送られるリクエストは道路の上を走る車と見なすことができます。車線が１つであれば、同時に1台の車しか通ることができません。多くの車が通ろうとすると、渋滞が起こってしまいます。多くの車をスムーズに通そうとするなら、車線を増やせばいいわけで

す。つまり、複数のWebアプリケーションが同時に動き、多くのリクエストを同時に処理できるようになれば、たくさんの仕事をこなすことができるようになるわけです。このように、1つのプログラムが同時に複数動くことを**マルチスレッドで動く**と呼ぶことがあります。

図03 多くのWebアプリケーションでは、複数のプログラムが同時に動く



Webアプリケーションがマルチスレッドで動いているとき、pickleを使ってデータを保存するとします。Pythonのオブジェクトをシリアライズして文字列に変換し、ファイルに書き込むわけです。

並行して動いているプログラムが、同じファイルに読み書きをするとどのようなことが起こるでしょうか。

並行して動いているプログラムが、1つのファイルを読み込もうとするときは、問題は起きません。しかし、1つのファイルに複数のプログラムが同時に書き込みを行おうとすると、問題が起こることがあります。平行して動いているプログラムが同時に1つのファイルに書き込みをしようとするわけですから、部分的に書き込む内容が重複してしまったり、不正な文字列が書き込まれてしまうかもしれません。Pythonのオブジェクトをシリアライズした内容が正しく書き出されていないと、オブジェクトの復元も正しく行われません。オブジェクトが復元されなければ、プログラムは当然正しく動かなくなってしまいます。

このようなトラブルを避けるためには、1つのファイルに対して同時に書き込みが起こらないように、書き込みの処理をブロックする必要があります。

同時書き込みをブロックするプログラムを書くのは意外と面倒で、また高度な知識が必要です。

そこで、たいていのWebアプリケーションでは、データを保存するために別の方法を活用します。それが**データベース**です。多くのデータベースでは、同時に書き込みが起こってもデータが壊れないように設計されています。Webアプリケーションの側では、同時に書き込みが起こっているかどうかを気にすることなく、データを保存できるわけです。

07-02 データベース概論

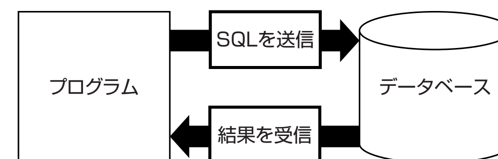
Webアプリケーションでデータを保存するためによく利用されるのがデータベースです。データベースを使うと、大量のデータを保存したり、保存したデータを素早く取り出すことができます。

リレーショナルデータベースとは

データベースにはいくつかの種類がありますが、中でもよく使われるのは**リレーショナルデータベース**（Relational Database）と呼ばれるデータベースです。リレーショナルデータベースでは、データを表のように見立てて登録、管理します。異なった種類の表のデータ間に**関連性**（Relation）を持たせることで、データ同士につながりを持たせ、複雑なデータを管理できるように設計されています。

たいていのデータベースは、独立したアプリケーションとして動きます。Webアプリのようなプログラムは、必要に応じてデータベースと通信を行います。通信を通して、データを保存したり、保存したデータを取り出したりするわけです。

図04 データベースとSQL



データベースは、Webアプリだけでなくさまざまなアプリケーションやプログラムが利用します。データベースにデータを保存したり、データを取り出したりするためには**SQL**（問い合わせ言語）と呼ばれる一種の言語を使います。SQLは言語ですので、Pythonと同じように文法があります。また、SQLには「標準」が定められています。多少の方言はあるものの、リレーショナルデータベースであればたいてい、標準的なSQLを利用することができます。

以下がSQLの簡単な例です。この例では、条件を設定してデータベースから情報を取り出しています。

SQLの簡単な例：

```
SELECT id, title FROM rss WHERE date > '2005-10-23';
```

「WHERE」という文字列の後にあるのが日付の条件です。日付は、西暦などの要素をハイフンで区切った文字列として与えられていることから分かるとおり、データベースとのやりとりに使われるSQL自体も文字列です。データベースを利用するプログラムでは、SQLを文字列として組み立てて問い合わせをし、データを保存したり、保存したデータを取り出すわけです。

リレーショナルデータベースには、無料で利用できるオープンソースのデータベースから、製品となっているものまでさまざまな種類があります。Pythonのようなプログラムからデータベースを利用するためには、専用のモジュールが必要です。Pythonには、世の中にある主要なデータベースに接続するためのモジュールが多く揃っています。

Python 2.5からは**SQLite**というデータベースに接続するためのモジュールが標準モジュールに内蔵されるようになりました。SQLiteは、データベース用のアプリケーションを使うことなく利用できる手軽なリレーショナルデータベースです。Python 2.5以上を使っていれば、Pythonを起動するだけでSQLiteを利用できるのです。本書では、このSQLiteを活用してWebアプリケーションを書きます。

ここからは、簡単にデータベースについて解説をしていきましょう。

テーブルとカラム

リレーショナルデータベースは**表**のような形式でデータを保存します。どのようなデータを保存するかによって、表の具体的な形式は変わります。リレーショナルデータベースでは、保存するデータの形式に合わせ、あらかじめ表の形式を定義しておきます。

この表の形式のことを**テーブル**と呼びます。区別するためにテーブルには名前を付けます。Pythonで新しい種類のデータ型を作りたいときにクラスを定義しますが、データベースのテーブルはそれに似ています。クラスにクラス名を付けるように、テーブルにはテーブル名を付けるわけです。

テーブルには保存したいデータにはどのような種類のデータが必要であるかを指定します。たとえば、住所を保存するテーブルを考えるとしましょう。「東京都千代田区何々」というように１つの文字列として住所を保存することも可能ですが、分類のためにいくつかの部分に分けるようにする方がよいでしょう。「郵便番号」、「都道府県」、「市区町村以下」というように3つの部分に分けるとしましょう。それぞれのデータを保存するための領域が表に必要になります。

この領域を**カラム**（Column）と呼びます。カラムには「カラム名」を付けます。Pythonのクラスに例えると、データを保存するためのアトリビュートがカラムに相当します。

また、実際にデータを保存すると、表に行が追加されていきます。これを、データベースでは**行**（Row）と言います。データベースの行は、Pythonのクラスインスタンスのようなものと考えてください。

図05 データベースでよく使われる用語の定義

	郵便番号	都道府県	市区町村
	100	東京都	千代田区○○△△
	001	北海道	札幌市○○△△
	100	東京都	千代田区○○△△

データベースでテーブルを定義するには、**CREATE TABLE**というSQL文を使います。以下のようなSQLを使ってテーブルを定義します。「CREATE TABLE」の後にある「address」というのは**テーブル名**です。

カラム名と、データの種類（データ型）を括弧で囲んで列記し、テーブルを定義します。idやprefectureという文字列がカラム名となります。

CREATE TABLE文の例：

```
CREATE TABLE address (
    postcode text,
    prefecture text,
    address text );
```

カラムとデータ型

テーブル定義を行うSQL文では、カラムの名前に添えて**データ型**を指定します。数値を保存したいときには数値型を指定します。同じ数値でも、整数を保存する場合と、小数点を含む実数を保存する場合では利用するデータ型が異なります。他にも、文字列を保存するためのデータ型、日付を保存するためのデータ型などが用意されています。

先ほどの例では、すべてのデータを文字列（text）として保存するようなテーブルを定義していました。

Pythonでもデータの性質によって利用するデータ型を変えますが、データベースでも同じことです。データベースの最大の特徴は、一度決めたデータ型は簡単に変更することができない、という点です。データベースによっては、あらかじめ定義したカラムのデータ型を変更できるものもありますが、その場合は保存されている列全体のデータ型を変換することになります。いずれにしても、カラムのデータ型はあらかじめしっかり決めておく必要がある、ということに変わりはありません。

IDとシリアル型

テーブル定義に利用するカラムとデータ型にはいくつか種類があります。中でも、よく利用されるカラムに**id**というものがあります。テーブルにたくさんの列が登録されているとき、特定の列を指定するために通し番号のようなものがあると便利です。idというカラムは、列につける通し番号としてよく使われます。idとは**識別子**（identifier）の頭文字を取ったものです。通し番号を指定すると、列が識別できる、というわけです。

idは通し番号ですので、データ型としては数値型を使えばよいことになります。新しい列が追加されたら、1つずつ増えてゆく番号、という特殊なデータ型としてシリアル（serial）型がよく利用されます。

先ほどの住所テーブルにシリアル型としてidというカラムを追加してみましょう。

idカラムを作成する例：

```
CREATE TABLE address (
    id serial,
    postcode text,
    prefecture text,
    street text );
```

SQL：データベース用の問い合わせ言語

SQLとはデータベースとの通信を行うための手法です。SQLはPythonと同じ言語の一種です。SQLにも文法があります。文法に沿ってSQL文字列として組み立てて、データベースに指示を与えます。

これまで、データを保存するための定義テーブルを作るSQLについて簡単に解説しました。他にも、データを登録したり、登録したデータを特定の条件に従って取り出す機能を持っています。

ここでは、よく利用されるSQLについて簡単に解説をしましょう。

INSERT文（データの登録）

あらかじめ定義してあるテーブルにデータを登録するために使うのがINSERT文です。テーブルに対してデータの挿入（insert）を行います。INTOの後にテーブル名を指定します。その後、データの挿入を行うカラム名と、VALUESに続けて挿入するデータを記述します。Pythonと同じように、SQLでは文字列を二重引用符で囲みます。シリアル型のidのようなカラムは、データが自動的に挿入されるので、INSERT文で指定することはありません。

INSERT文の例：

```
INSERT INTO address (postcode, prefecture, street)
       VALUES("100-0000", "東京都", "千代田区1-1");
```

データベースのテーブルをPythonのリストのリストだとすると、INSERT文はリストに対して要素を追加するappend()メソッドに似ています。

SELECT文（データの選択）

テーブルに登録されているデータを取り出すために利用するのがSELECT文です。SELECT文の後にデータを取り出すカラム名を指定します。その後、データを取り出すテーブル名を指定します。

SELECT文の例：

```
SELECT prefecture, street FROM address
    WHERE postcode="100-0000";
```

WHERE句以降ではデータを取り出す条件を指定します。ここでは、addressテーブルのpostcodeカラムを指定して、郵便番号で絞り込みを行っています。WHERE句には、ANDやORを使った複合条件を指定することもできます。WHERE句を指定しないと、すべてのデータを選択することになります。

WHERE句の他にも、取り出すデータの数を制限する**LIMIT句**や並び順を指定する**ORDER BY句**などをあわせて利用することができます。

SELECT文はPythonのリストのリストから値を取り出す**リスト内包表記**に似た働きを持っていると言えます。

UPDATE文（データの更新）

テーブルに登録されているデータを更新するために利用するのがUPDATE文です。UPDATE文の後にテーブル名を指定し、SET句の後に更新をするカラム名と値をイコールでつなげて指定します。更新する値が複数ある場合はカンマで区切って列記します。

INSERT文の例：

```
UPDATE address SET postcode="001-0000", prefecture="北海道"
    WHERE id=10;
```


WHERE句はSELECT文でも出てきましたが、UPDATE文でも更新対象のデータを指定するために利用します。条件に該当するデータが複数あった場合には、複数の列が更新されます。この例では、idを指定して1つの列だけを更新しています。

DELETE文（データの削除）

登録済みのデータを削除するために利用するのがDELETE文です。SELECT文、UPDATE文と同じく、削除対象となるデータの条件をWHERE句で指定します。この例では、idを指定して1つのデータを削除しています。

DELETE文の例：

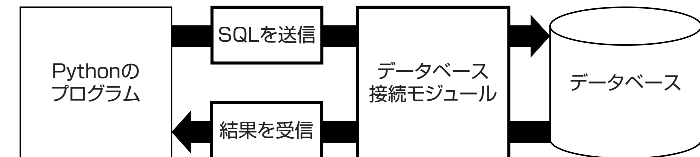
```
DELETE FROM address WHERE id=11;
```

WHERE句を指定しないこともできますが、その場合はテーブル上の全データを対象に削除を行うことになります。

07-03　Pythonとデータベースの連携

データベースは、Pythonとは違った場所で動いている独立したアプリケーションであることがほとんどです。Pythonからデータベースを使うには、データベース接続用の拡張モジュールを利用します。この拡張モジュールを使ってデータベースに接続し、SQLなどを利用してデータベースと通信をするわけです。

図06　Pythonとデータベース接続モジュール



Pythonには、よく利用される主要なデータベースに接続するための拡張モ

ジュールが一通り揃っています。Python 2.5からは、SQLiteというデータベースと連携をするためのモジュールが標準で搭載されています。SQLiteは、独立したアプリケーションとして動作しないタイプのデータベースです。Pythonを起動するだけで、手軽に本格的なデータベースの機能が利用できます。

ここでは、SQLiteを利用してPythonとデータベースの接続の具体例を見てみましょう。

SQLiteをPythonから利用する

SQLiteはとても手軽で高機能なデータベースシステムです。データベースを使うに当たってデータベースサーバなどを専用に立ち上げる必要はありません。PythonからSQLiteを使うには、sqlite3というライブラリがあればOKです。

Python 2.5からは、sqlite3が標準ライブラリに含まれています。つまり、Pythonをインストールするだけでデータベースが利用できるわけです。Python 2.5以前でSQLiteを使うには、pysqliteというモジュールをインストールする必要があります。

SQLiteはデータベースに登録されたデータをファイルに書き出します。ライブラリを使うときに、データを保存するファイルのパスを指定する必要があります。同じファイルを使えばデータベースに登録されたデータの内容を復元することができますので、PythonやPythonを使ったアプリケーションを終了しても、データは残っています。

PythonとSQLiteを接続する

PythonからSQLiteに接続するためには、まずモジュールをインポートする必要があります。その後、モジュールの関数を使って**コネクションオブジェクト**と呼ばれるオブジェクトを作ります。コネクションオブジェクトには、接続文字列と呼ばれる文字列を引数に渡して作ります。接続文字列は接続するデータベースによって異なりますが、SQLiteの場合はデータを保存するファイルのパスを指定することになります。

コネクションオブジェクトを作ったら、次は**カーソル**と呼ばれるオブジェクトを作ります。データベースとのやりとりはカーソルを経由して行うことになります。SQLをデータベースに送ったり、SQLを実行した結果を受け取るときにカーソルを使うわけです。

以下のサンプルでは、コネクションオブジェクトとカーソルオブジェクトを生成し、SQLiteにSQLを送信しています。

PythonからSQLiteに接続してSQLを送信

```
>>> import sqlite3
>>> con=sqlite3.connect(":memory")
>>> cur=con.cursor()
>>> cur.execute("""CREATE TABLE address(
... id serial, postcode text,
... prefecture text, street text);""")
<sqlite3.Cursor object at 0x7ccb0>
>>> cur.execute("""INSERT INTO address(postcode, prefecture,
street)
... VALUES('100', 'Tokyo', 'Chiyodaku');""")
<sqlite3.Cursor object at 0x7ccb0>
```

DBAPIを使う

Pythonでは、いろいろなデータベースと接続するための共通の仕組みとしてDBAPI 2.0と呼ばれる手法が利用されています。Pythonには、いろいろなデータベースに接続するためのモジュールが用意されています。しかし、接続や通信の方法はモジュールごとにまちまちであることがほとんどです。通信や接続の方法をある程度統一し、Pythonからさまざまなデータベースをより手軽に利用できるようにする目的もあり、DBAPIが作られました。個別のモジュールの利用方法を知らなくても、DBAPIの利用方法だけを知っていれば、Pythonからデータベースを利用できる、というわけです。

DBAPI 2.0では、コネクションオブジェクトと**カーソル**という2つのオブジェクトを使い、データベースとの通信を行います。

●コネクションオブジェクト

データベースとの接続を行うときに利用するオブジェクトです。データベース接続モジュールの関数を呼び出して作成します。コネクションオブジェクトを作る関数には、引数として**接続文字列**と呼ばれる文字列を渡します。接続文字列には、接続するデータベースに関する情報を書き込みます。接続

文字列としてどのような文字列を渡すかは、接続するデータベースによって異なります。

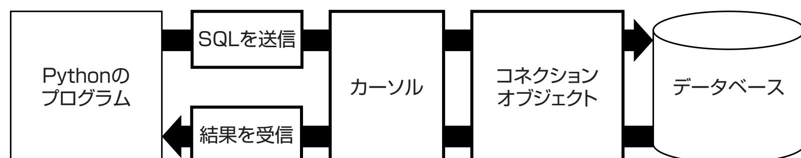
データベースの操作が終わり、接続を切るためにもコネクションオブジェクトを使います。

● カーソル

データベースにSQLを送信して問い合わせを行ったり、結果を得るために利用するオブジェクトです。コネクションオブジェクトのメソッドを呼び出して作ります。

データベースに問い合わせを行うためには、カーソルのメソッドにSQL文字列を渡します。データベースからデータを取得するような問い合わせを行った場合や、データベースから返ってきたデータを取得するためにもカーソルを利用します。問い合わせを行った後、カーソルに対してメソッドを呼び出すことで、データを取得します。

図07 コネクションオブジェクトとカーソル



コネクションオブジェクトを作る

データベースに対する操作を行うためには、まずコネクションオブジェクトを作ります。SQLite用のコネクションオブジェクトを作るには、sqlite3モジュールのconnect()関数を呼びます。

● connect (database (接続文字列))

connect()関数には、SQLiteがデータベースの内容を保存するファイルのパスをフルパスか相対パスで指定します。ファイルを変えることで、複数のデータベースを利用することができます。

":memory:"という文字列を指定すると、ファイルを使わず、メモリ上にSQLiteデータベースを保存します。ただし、メモリ上に保存したデータは接続を閉じると消えてしまいます。開発中やテストを行うときに利用すると便利です。

データベースとの接続を切断するには、コネクションオブジェクトに対してclose()メソッドを送信します。

カーソルを使う

SQLを使ってデータベースと通信を行うためにはカーソルを作ります。カーソルは、コネクションオブジェクトのメソッドを呼ぶことで作ります。

● cursor ()

カーソルオブジェクトを返します。このカーソルオブジェクトを使うことで、コネクションオブジェクトに指定されたデータベースに対して通信を行います。

カーソルオブジェクトでは以下のメソッドが利用できます。まずexecute()メソッドでSQLを発行して、必要に応じてfetchone()やfetchall()を使ってデータを取り出す、という処理の流れになります。

● execute (sql (SQL文字列) [, パラメータ])

SQLを引数として渡し、実行します。

完全なSQLを文字列として渡すほか、**プレースホルダ**と呼ばれる置換用の文字列を含んだSQLのテンプレートを利用できます。プレースホルダを使うと、一部分だけ変化するSQL文字列を効率的に処理できます。

ただし、execute()メソッドを実行しただけでは、insert文やupdate文によるデータベースの更新は行われないので注意してください。PythonのDBAPIでは**トランザクション**機能が自動的にオンになるためです。本書ではトランザクションについて詳しく解説しませんが、簡単に言うと、複数のデータベース更新処理が重なったときに、データベースに矛盾が起きないように排他制御をするための機能です。

データベースに対する更新を有効にするために，コネクションオブジェクトの**commit()**というメソッドを呼ぶ必要があります。commit()メソッドを呼ぶと，トランザクションが閉じられます。commit()メソッドを呼ぶ前にカーソルオブジェクトが消滅すると、トランザクションがロールバックされてし

まい，更新の内容がキャンセルされてしまいます。

なお、明示的にトランザクションをロールバックしたいときには，**rollback()**メソッドを呼びます。

● **fetchone ()**

カーソルから選択したデータを1行だけ取り出します。戻り値は、選択した結果を順番に格納したタプルとなります。データがない場合には**None**が返ってきます。SQLを使ってデータベースからデータを選択した後で呼び出します。

● **fetchall ()**

カーソルから選択したデータをすべて取り出します。戻り値は、選択した結果を順番に格納したタプルのタプルとなります。

また、選択を行った後のカーソルオブジェクトをfor文に添えると、このメソッドが呼び出されます。選択結果を1行づつ繰り返し変数に代入して処理を行えるわけです。

● **close ()**

カーソルを閉じます。close()メソッドを呼んだ後のカーソルでexecute()メソッドなどを呼び出そうとするとエラーになります。

SQLの動的生成

Webアプリケーションなどでは、SQLに指定する値だけを変えて処理をする、というようなことを頻繁に行います。たとえば、データを登録するINSERT文では、VALUES以下の値だけが異なり、その他の文字列はいつも決まっています。SQLを文字列として見ると，固定している部分と変化する部分があるわけです。

Pythonを使って、先ほどのaddressテーブルにデータを登録する方法を考えてみましょう。郵便番号、都道府県などの情報は変数に入っているとします。変数を使ってSQLに相当する文字列を組み立て、カーソルオブジェクトのexecute()メソッドを使ってデータベースに渡すわけです。

変数を使って文字列を組み立てる方法は何種類か考えられます。一番素朴な方法は文字列の足し算を使う方法でしょう。postcodeという変数に郵便番号が数値として入っていて、prefectureという変数に都道府県が文字列として入っているとします。INSERT文を組み立てるためには以下のようにすればいいはずです。

```
sql="INSERT INTO address (postcode, prefecuture) VALUES ("
sql+="'"+str(postcode)+"','"+prefecuture+"');"
```

SQLでは、文字列を引用符で囲む必要があります。文字列と変数の足し算をして、文字列をクオーテーションで囲むようなSQL文字列を作るようにするわけです。処理としては単純ですが、スマートな書き方とはいえません。また、クオーテーションが入り組んでいるので、修正がとても大変そうです。INSERT文を文字列として見ると、処理内容によって変化する場所はいつも同じです。固定した部分と変化する部分を分け、変化する部分だけを効率的に書き換える方法があれば、もっと手軽にSQLを扱えるはずです。

プレースホルダを使う

カーソルオブジェクトのexecute()メソッドにSQLを渡すには2種類の方法があります。１つは完全なSQL文字列を渡す方法。そしてもう１つはプレースホルダを使う方法です。プレースホルダは一種のテンプレートのように使えます。SQLのうち定型の部分と処理内容によって変化する部分を効率的に扱えるわけです。

プレースホルダをSQLに埋め込むには2種類の方法があります。

１つ目の方法はクエスチョンマークを使う方法です。オプションの引数にタプルを渡すと、クエスチョンマークのある場所にオブジェクトが順番に埋め込まれます。

次の例では、INSERT文の条件の部分をプレースホルダを使って書き換えています。「name」、「age」という変数にそれぞれ「guido」（文字列）、「23」（数値）という値が代入された状態で以下のようなメソッドを実行するとします。

```
cur.execute("""SELECT lastname FROM people
            WHERE firstname=? AND age=?""", (name,age))
```

結果として、以下のようなSQLがデータベースに送信されます。

```
SELECT lastname FROM people WEHRE firstname='guido' AND age=23
```

もう１つの方法は、辞書のキーを使って置換する場所を指定する方法です。クエスチョンマークを使った方法と違って、引数の順番の影響を受けないのが便利な点です。

代入された状態で以下のようなメソッドを実行すると、前の例と同じSQLが送信されます。

```
cur.execute("""SELECT lastname FROM people
               WHERE firstname=:name AND age=:age""",
            {'name':name,'age':age})
```

プレースホルダは、文字列をクエリとして記述するときに必要なクオーテーションやエスケープ処理などを自動的に行います。Webアプリケーションなどでパラメータに SQLを挿入することで起こる**SQLインジェクション**というセキュリティ上の問題も起こりにくくなります。文字列を連結してSQLを作るより、手軽で安全な手法と言えます。

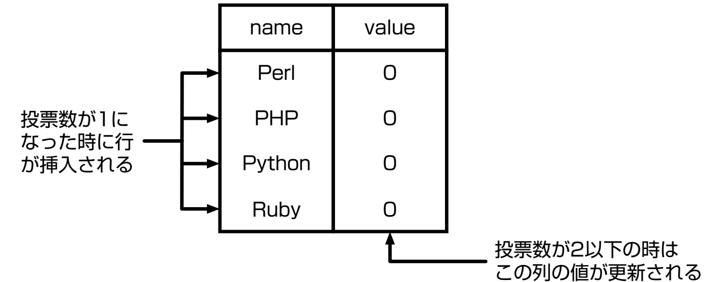
Webアプリケーションでデータを保存する - データベース編

Pythonからデータベースを利用する方法を一通り学びました。ここでは、データベースを利用してデータの保存を行うWebアプリケーションを作ってみましょう。先ほど作った投票Webアプリケーションを、データベースにデータを保存するように改造してみます。

データベース操作は、DBAPIを利用します。まず、sqlite3モジュールを使ってコネクションオブジェクトを作ります。コネクションオブジェクトからカーソルオブジェクトを作り、SQLを文字列として組み立て、データベースとの通信をする、という手順になります。

データベースにデータを保存するためには、テーブルを定義する必要があります。今回作成するWebアプリケーションでは、「言語名」と「投票数」の2つのデータを管理します。CREATE TABLE文を使って、この2つのフィールドを持ったテーブルを定義することになります。

図08　テーブル（language_pole）の構造



まず、言語ごとにINSERT文を使ってテーブルに行を追加します。投票が行われるごとに、UPDATE文を使って投票数を加算していけば、投票結果を保存できるはずです。

前回のプログラムを元に、データを登録、保存する部分のコードを書き換えます。投稿用のフォーム、結果を表示するグラフを表示するためのHTMLを作るコードは、前に作ったものからほぼそのまま流用できます。

以下がデータベース版の投票Webアプリです。データベースとの通信部分が加わったことで、プログラムが少し長くなっています。他のプログラムと同様に、実行権限を与えるなどしてcgi-binフォルダにファイルを置いてください。

List02　dbpole.py

```
#!/usr/bin/env python
# coding: utf-8

import sqlite3
from httphandler import Request, Response, get_htmltemplate
import cgitb; cgitb.enable()

form_body=u"""
  <form method="POST" action="/cgi-bin/dbpole.py">
    好きな軽量言語は?<br />
    %s
    <input type="submit" />
  </form>"""

radio_parts=u"""
```

```
<input type="radio" name="language" value="%s" />%s
<div style="border-left: solid %sem red; ">%s</div>
"""

def incrementvalue(cur, lang_name):
    cur.execute("""SELECT value FROM language_pole
                  WHERE name='%s'""" % lang_name)
    item=None
    for item in cur.fetchall():
        cur.execute("""UPDATE language_pole
           SET value=%d WHERE name='%s'""" % (item[0]+1,
lang_name))
    if not item:
        cur.execute("""INSERT INTO language_pole(name, value)
                      VALUES('%s', 1)""" % lang_name)
con=sqlite3.connect('./dbfile.dat')                          ❶
cur=con.cursor()
try:                                                         ❷
    cur.execute("""CREATE TABLE language_pole (
                  name text, value int);""")
except:
    pass

content=""                                                   ❸
req=Request()
if req.form.has_key('language'):
    incrementvalue(cur, req.form['language'].value)

lang_dic={}                                                  ❹
cur.execute("""SELECT name, value FROM language_pole;""")
for res in cur.fetchall():
    lang_dic[res[0]]=res[1]

for lang in ['Perl', 'PHP', 'Python', 'Ruby']:
    num=lang_dic.get(lang, 0)
    content+=radio_parts%(lang, lang, num, num)

con.commit()

res=Response()
body=form_body%content
res.set_body(get_htmltemplate()%body)
print res
```

プログラムの前半部分は、モジュールインポート、テンプレート文字列や関数定義などを行う部分です。後半部分からが、Webアプリケーションの処理をしている部分になります。

まず、sqlite3モジュールを使ってコネクションオブジェクトを作っています。コネクションオブジェクトを作るときにはSQLiteがデータを保存する際に利用するファイルを指定しています。直後では、コネクションオブジェクトからカーソルオブジェクトを作っています（❶）。

その次のコードでは、データを保存するテーブルを作っています。カーソルオブジェクトを使ってCREATE TABLE文を文字列としてデータベースに送ることで、データベースにテーブルを作るよう指示をしています。

「language_pole」という名前のテーブルは1回しか作れません。2回目以降作ろうとするとエラーが発生します。このプログラムでは、try〜except文でコードを囲み、エラーが起こってもそのままプログラムを実行するようにしています（❷）。

次に、リクエストとして送られてきた投票の内容をデータベースに記録するコードが続きます。リクエストの中のクエリを調べ、投稿内容をデータベースに保存するincrementvalue()関数（❸）を呼び出しています。

incrementvalue()関数では、SELECT文、INSERT文、UPDATE文の3種類のSQLを組み合わせて投票内容のアップデートをしています。まずSELECT文を使って、ある言語名を保存した列があるかどうか調べています。もし列がない場合は、新しく列を追加する必要があるので、INSERT文を使って列を追加しています。すでに列がある場合は、UPDATE文を使って列の内容を更新しています。

その次には、データベースにSQLを送り、これまでの投稿データを取り出しています。データベースから登録済みのデータを取り出すためにはSELECT文を使います。language_poleに登録されたデータをすべて取り出し、結果を使ってループを組んでいます。ループの中では、言語名をキーにした辞書を作っています（❹）。

最後に、投票結果を元にして、投票用のフォームと結果の棒グラフを作り、レスポンスとして返しています。

Part 2

効率的なWeb
アプリケーション開発

CHAPTER 08

効率的なWeb アプリケーション開発とは

これまでの章で、Webアプリケーション開発に必要な技術のうち基本的な部分を解説しました。この章では、より高度なWebアプリケーションを開発する上で、必要になる技術や手法について解説します。

Webアプリケーションの基本は**文字列処理**です。Webアプリケーションのプログラムでは、文字列の追加や置換、データ型の変換といった文字列処理の手法をたくさん使います。

アプリケーションにとって重要な要素である**入力**は、WebアプリケーションではWebブラウザなどから送られてくる**リクエスト**上にクエリ文字列として渡されます。クエリ文字列を分割してデータを取り出し、アプリケーションと本体となるプログラムの内部で処理をします。

また、Webアプリケーションの**出力**でもHTMLという文字列を使います。Webアプリケーションでは、処理の結果をHTMLを使って表示します。Webアプリケーションの内部では、処理の結果としてユーザに見せるHTMLを文字列として組み立て、**レスポンス**として返します。

Webアプリケーションでデータを保存するためにはデータベースをよく利用します。データベースからデータを取り出したりデータを保存するためには、SQLを使ってデータベースと通信を行います。このSQLも文字列の一種です。取り出したいデータの条件や、保存したいデータをSQL文字列として組み立て、データベースに対して指示を与えます。

このように、Webアプリケーションでは多くの部分で文字列処理を行います。文字をつぎはぎする処理を積み重ねてゆくことで、原理的には、どんなに複雑なWebアプリケーションでも作り上げることができるはずです。

08-01　アプリケーションと開発効率

プログラムとは、処理を手順に沿って並べた文字列にすぎません。これは、ファイルの行数を数えるような単純なプログラムであっても、飛行機の運航を管理するような複雑なプログラムであっても同じことです。すべてのプログラムは、あらかじめ決められた手続きを、プログラムに書かれた手順に沿って実行しているにすぎません。

Webアプリケーションの基本はテキスト処理です。Webアプリケーションはとてもシンプルな原理で動いているため、単純な処理の積み重ねであらゆることができるように思えてきます。

Webアプリケーションで実行する処理の多くは文字列の切り貼りです。出力に使うHTMLもデータベースとの通信に使うSQLも、全部1つのプログラムの中に埋め込んでしまう方が便利に思えます。事実、初期のWebアプリケーションのほとんどはそのような素朴な手法で作られていました。単純な処理をするWebアプリケーションでは、そのような素朴な手法がうまく機能しました。Pythonのようなスクリプト言語には、文字列を扱うための強力な機能が多く搭載されています。また、データベースとの接続など、豊富な機能を提供する標準モジュールを使えば、とても手軽にWebアプリケーションを作ることができます。

素朴な手法の問題点

Webアプリケーションのプログラムで必要となる文字列の切り貼りを、その場で順番に行う。これをここでは**素朴な手法**という言葉で呼びましょう。これまでのサンプルプログラムでは、このような手法を使ってプログラムを作ってきました。最初に作った、単純な出力だけを行うWebアプリケーションでは特に問題があるようには見えません。

サンプルプログラムの後半になると、だんだんとプログラムの見通しが悪くなってきています。UIとなるフォームをWebアプリケーションから出力したり、データベースとの通信を行い始めると、Pythonのコード以外の文字列がたくさんプログラムに入り込んでしまっています。

このようなプログラムではプログラムの変更をするとき、変更すべき場所を探し出すのが大変になります。Webアプリケーションに機能を追加したい

とき、不具合を直したいときは、HTMLやSQLのような文字列をかき分けて、該当するPythonのコードをいちいち見つけ出さなければなりません。

また、コードだけが修正されるとは限りません。今の状態でさえ、いろいろな種類の文字列が交じってしまっています。Webアプリケーションをもっと凝ったデザインするときや、専門のデザイナーにデザインを頼むときに、どのようなことが起こるか想像してみるとよいでしょう。

Webアプリケーションの歴史の初めのころは、素朴な手法を使って開発が行われていました。しかし、このような手法も程なく壁に行き当たります。Webアプリケーションに対する要求が大きくなり、より複雑な機能を持ったWebアプリケーションを作る必要が出てきたのです。そうなると、不具合の修正や機能追加がだんだんと行いづらくなり、うまく開発が進まなくなってきたのです。

モジュール化、クラス化による開発の効率化

素朴な手法を使ったプログラミングでは、文字列処理などを積み重ねていくことでプログラミングを行います。これはちょうど、レンガを積み上げて建築をするようなものです。レンガを積み上げていけば、小屋や小さな家くらいは簡単に作れるかもしれません。しかし、巨大なビルを作るには、レンガを積むような手法を使うのではなく、もっと効率の良い、他の手法が必要です。

Webアプリケーションに限らず、プログラムには多くの「似通った処理」があります。似通った処理の性質をよく分析して、処理をまとめることによって、似通った処理を繰り返しコードとして書かなくてよいことになります。Pythonでは、そのような処理を**モジュール**や**クラス**としてまとめることが多いはずです。

Webアプリケーションで扱うテキストはたいてい一定のルールを持っています。たとえば、Webアプリケーションの出力となるレスポンスには**ヘッダ**と**本文**という2つの部分があります。ヘッダも本文もどちらも文字列です。本文の文字列は、どのような結果をユーザに見せたいかによって大きく変化します。ヘッダには表面に現れないいろいろな情報を記載しますが、記載する情報の多くは**おのずと決まる**ような値が多いため、プログラムで自動的に生成することができます。

このような性質に注目して、Responseという名前のクラスを作りました。レスポンスを共通して扱うクラスを作ったことで、プログラムがよりシンプルになりました。Webアプリケーションで実行する共通した処理をクラスとしてまとめたことで、プログラムに重複部分がなくなり、プログラム全体がスッキリと短くなったわけです。

レスポンスだけでなく、データベースと通信を行うときに利用するSQLもルールを持った文字列です。SQL文字列を作るときのルールをうまく分析し、クラスにすることができればどうなるでしょうか。プログラムに埋め込まれたSQL文字列がきれいになくなり、プログラムはさらにスッキリするはずです。実際、近年のWebアプリケーションの開発では**O/Rマッパー**といわれるクラスの一種が活用されています。O/Rマッパーについては、この章でのちほど詳しく解説します。

役割分担による開発の効率化

Webアプリケーションの開発の中で最もやっかいなのがHTMLに相当する文字列です。HTMLには、数値や文字列といった情報だけでなく、テキストの大きさや位置、テキストの意味などを決める情報をタグ（エレメント）という形で記述します。このため、一般的なテキストに比べてHTMLは長くなりがちです。横幅が長くなるのも、Pythonのプログラムとしてはあまり嬉しくありません。

このような長いテキストを埋め込むと、プログラムはすぐにHTMLで埋め尽くされてしまいます。そのため、Webアプリケーションの出力として使うHTMLだけを外部に分離しておく、という手法がとられ始めました。プログラムとは別のファイルにHTML文字列を書いておき、必要に応じて読み込んで使うのです。このような仕組みは**テンプレートエンジン**と呼ばれています。高機能なテンプレートエンジンでは、単にHTMLファイルを読み込むだけでなく、内部にプログラムコードのようなものを埋め込めるものもあります。

HTML文字列をプログラムから外部に追い出すことによって得られる恩恵はいくつかあります。まず、長くなりがちなHTMLがプログラム上になくなることによって、プログラムがスッキリすることが一点。もう一点は、役割分担が明確になる、という点です。

たとえば、処理の結果として10個の数値を表示するWebアプリケーション

を作るとします。10個の数値の表示方法はいろいろあります。横に並べるか、縦に並べるか。文字の大きさはどうするか。Webアプリケーションを作っている過程で、いろいろなパターンを試し、最も見やすい方法で表示することになるでしょう。

もしHTMLがプログラムに埋め込まれていたら、表示方法を変えるたびにプログラムの修正が必要になります。毎回、プログラムに埋め込まれているHTMLを修正するわけです。一方、10個の数字を作るという、Webアプリケーションの根本とも言える処理の部分は変更する必要がありません。一方、Webアプリケーションの出力として使うHTMLを分離しておけば、表示方法を変更したくなったら分離したHTMLだけを修正すればよくなります。

Webアプリケーションには**データを作ったり処理する機能**と**データを表示する機能**という2つの大きな機能があります。このうちデータを表示する機能には頻繁に変更が加わります。頻繁に変更が加わる処理は分離しておいた方が効率的に開発ができます。Webアプリケーションの出力として使うHTMLを分離しておく、ということには、このような効能もあるのです。

近年のWebアプリケーションの開発では、このようにWebアプリケーションの処理内容を明確に分けて、役割分担をする、という手法が良く取られます。データを作ったり処理する機能はさらに2つに分けられることもあります。1つは、データベースとのやりとりが必要な部分です。もう1つは、リクエストを受け取ったり、データを加工する部分です。データベースに関わる部分をM（モデル）、表示に関わる部分はV（ビュー）、リクエストを受け取ったりデータを加工する部分をC（コントローラ）と呼んで分類することがあります。

素朴な手法を使っていたかつては、Webアプリケーションのすべての機能が数個のソースコードに雑然と並んでいました。近年のWebアプリケーションの開発では、プログラムの部品化を進め、役割分担を明確にするような手法がよく採用されます。そのような手法を使うことで、高度な機能を持ったWebアプリケーションをより素早く手軽に開発できるようになります。

この章では、近年よく利用される開発手法を具体的に見ていきながら、より高度なWebアプリケーションを効率よく開発するための方法について解説します。

CHAPTER 09

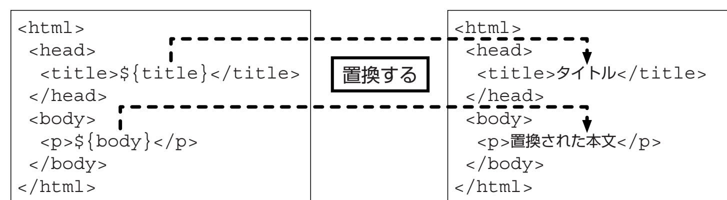
Pythonと テンプレートエンジン

テンプレート（template）とはもともと「型板」や「鋳型」という意味の英語です。そこから発して「定型書式」にもこの言葉が使われるようになりました。たとえば、ワープロにはたくさんの定型文が付属していることがあります。祝辞や季節の挨拶など、決まり切った形式のある文章のひな形があらかじめ用意されていることがあります。このような文書をテンプレートと呼ぶことがあります。テンプレートをひな形として、相手や送り主の名前、日付、季節など変化する部分だけを入れ替えて利用するのです。Eメールで決まった形式のある文書を送るときも、似たような機能を利用することがあります。

09-01 テンプレートエンジンとは

文字列やテキストを扱うときも、定型の部分と変化する部分をうまく切り分け、変化する部分だけを必要に応じて書き換えることができると便利です。定型の文字列に、置き換わる部分を埋め込み、必要に応じて出力を変化させるのです。そのようなときに利用するのが、**テンプレートエンジン**（またはテンプレートシステム）と呼ばれる仕組みです。

図01 テンプレート



Webアプリケーションの出力として利用するHTMLは定型の部分が多くあります。HTMLは多くの種類のタグを使って記述します。タグは表示をする文字や画像などの種類、位置、大きさを決める役割を持っています。このため、タグ自身はあまり変わることがありません。Webアプリケーションでよく変わるのは、タグとタグの間にある文字列です。つまり、Webアプリケーションの出力として使うHTMLは、定型の部分と変化する部分がはっきり分かれていることが多いのです。

これまで、本書で取り上げてきたサンプルプログラムでは、テンプレートエンジンを使いませんでした。そのため、HTMLがプログラムの中に紛れ込んでいました。

HTMLがプログラムの中に紛れ込んでいると、プログラムがとても見にくくなってしまうことがあります。プログラムコード自体も一種の文字列です。Webアプリケーションの動きや機能を制御するためのプログラムの主要な部分と、HTMLのようなプログラム以外の文字列が紛れて、どの部分がどのような処理をしているのかが分かりづらくなってしまいます。また、HTMLのような文字列がプログラムに紛れ込むことで、プログラム全体が無用に長くなってしまいます。

そもそもHTMLには、文字や画像の大きさなどを制御するための文字列が多く含まれています。たとえば、Webアプリケーションが出力する文字の大きさを変えたいという目的のためだけに、プログラムに埋め込まれたHTMLを変更する必要があるのでとても面倒です。

HTMLのように、Webアプリケーションの動作に直接関係のない情報は、プログラムと別に管理して**分離**をした方が便利です。Webアプリケーションの動作を変えたいときにはプログラムを修正すればよいですし、文字の大きさなどを変えたいときにはテンプレートを変更すればよいわけです。

Webアプリケーションを作るとき、テンプレートの機能を活用できれば、より効率的にアプリケーションの開発が行えるはずです。

09-02　テンプレートエンジンの動く仕組み

プログラムで扱うテンプレートにはたくさんの種類があります。Webアプリケーションで使われるテンプレートエンジンにはよく似た特徴があります。テンプレートエンジンの正体は、文字列の置換を行うことを目的に作られた専門のプログラムです。多くのテンプレートエンジンは関数やクラスとして実装されています。Webアプリケーションでは、テンプレートエンジンの機能を活用して、出力として利用するHTMLを作ります。

まず、多くのテンプレートエンジンは独自のパターンを持った記法を持っています。HTMLのような文字列の中に、特定のパターンを持った文字列を埋め込むことで、変化する文字列の場所を指定するわけです。

置き換えを行うために利用する**特定のパターン**の具体例を示してみましょう。たとえば「${〜}で囲まれた部分を置き換える」というような置換のためのルールがあるとします。以下の例だと、${title}という部分が他の文字列に置き換わって表示されます。もちろん、テンプレートエンジンの種類によって、置換ルールは異なります。

```
<h1> ${title} </h1>
```

文字列の中に変数を埋め込むようなイメージで、テンプレートの本体となる文字列を書いていきます。このようにして、Webアプリケーションの出力に利用するHTMLのひな形を作ります。ひな形は、独立したテキストファイルとして設置する場合もありますし、短いものならプログラムの中の文字列リテラルとして埋め込むこともあります。

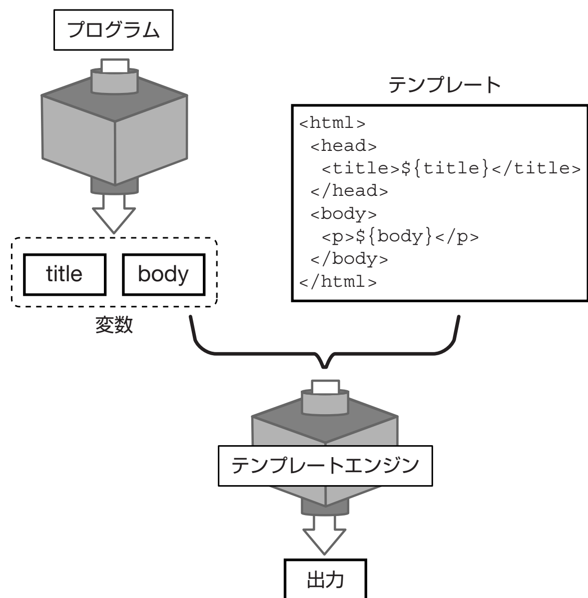
このようにして作ったテンプレートは、テンプレートエンジンの本体となるプログラムで読み込みます。テンプレートエンジンのプログラムでは、テンプレートに埋め込まれた置換ルールを探し出し、必要に応じて置換を行います。

Pythonで利用できるテンプレートエンジンの多くはクラスとして実装されています。まず、テンプレート文字列を指定するなどしてクラスのインスタンスオブジェクトを作ります。その後、テンプレートに埋め込まれたパターンを置換するためのメソッドを呼び出して、Webアプリケーションの出力となるHTMLのような文字列を作ります。

テンプレートエンジン側で置換を行う文字列のようなデータは、Webアプリケーションのプログラム本体で作ります。テンプレートエンジンがHTML

を出力するときに、プログラムで作った文字列などを渡します。テンプレートエンジン側では、受け渡された置換に必要な情報を元に、HTMLなどを生成して、出力します。Webアプリケーションのプログラムとテンプレートエンジンは、ちょうど次の図のような関係になっています。

図02　テンプレートエンジン



09-03　標準モジュールを使ったテンプレートエンジン

Pythonの標準モジュールには、簡単な機能を持ったテンプレートエンジンが内蔵されています。stringモジュールに含まれている**Templateクラス**です。ここでは、そのテンプレートエンジンの使い方を簡単に解説しましょう。

ところで、Pythonには**文字列テンプレート**と呼ばれる機能があります。文字列の中に**%s**、**%d**といった記号を埋め込んでおくと、動的に文字列の置換を行うことができます。その他に、文字列テンプレートでは、**%(key)s**のように辞書のキーを指定して置換を行うこともできます。Templateクラスは、この機能をより高度にしたクラスです。

Templateクラスの使い方

クラスですので、テンプレートの機能を使うためにはインスタンスオブジェクトを生成する必要があります。Templateクラスでは、インスタンスを生成するときに文字列を引数として取ります。以下のようにして、テンプレートのもととなる文字列を与えてインスタンスオブジェクトを作ります。以下の例では、ファイルに書かれている文字列を引数として渡し、Templateクラスのインスタンスオブジェクトを作っています。

```
t=Template(open('./tmpl.txt').read())
```

「tmpl.txt」というファイルには、以下のような文字列が定義されているとします。**${~}**という部分が置換される文字列になります。

```
<html><body>
<h1> ${title} </h1>
<p>${body}</p>
</body></html>
```

Templateクラスのインスタンスであるtを使い、置換結果の文字列を得るにはsubstitute()というメソッドを呼びます。このとき、メソッドの引数に置換用のパターンに埋め込みたい文字列を渡します。渡し方は2種類あります。1つは、置換したい文字列を納めた辞書を渡す方法です。もう1つは、キーワード引数として文字列を渡す方法です。

テンプレート上では、**title**と**body**という2種類のキーが渡ってくることが期待されています。WebアプリケーションなどからTemplateクラスを使うときには、辞書やキーワード引数の形で、データを受け渡すことになります。以下の2つの例は、どちらも同じ結果を返します。

```
t.substitute({'title':'The title', 'body':'This is body''})
```

```
t.substitute(title='The title', body='This is body''})
```

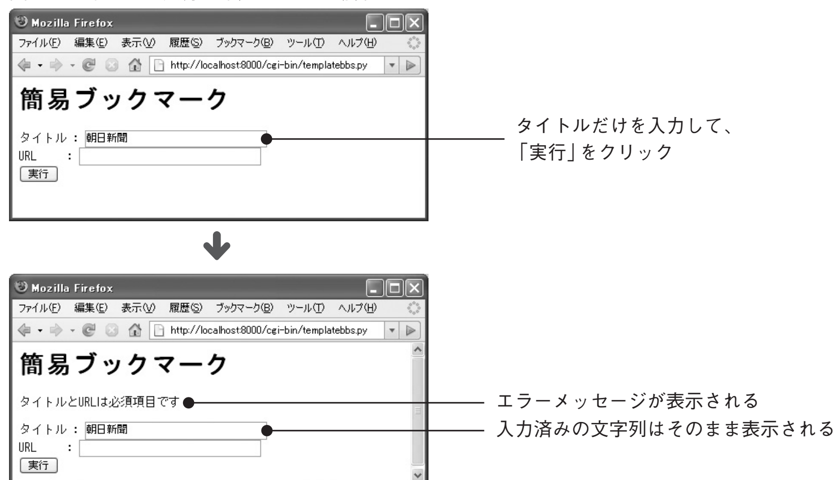
Templateクラスを活用してブックマーク管理Webアプリを作る

テンプレートエンジンを使うとWebアプリケーションの実装がどのように変わるのかを実感するために、Webアプリケーションを作ってみましょう。今回は、ブックマーク管理を登録するためのWebアプリケーションを作ってみます。フォームを使って、WebサイトなどのURLとタイトルを登録し、ブックマークを管理するためのWebアプリケーションを作ります。

ブックマークの要素を入力するUIとしてフォームを利用します。Webアプリケーションでフォームからの入力を受けて、データベースにデータを登録します。また、1つのプログラムで、フォームの表示とデータの登録を行うようなWebアプリケーションを作ることにします。

今回のWebアプリケーションでは、ちょっと凝った処理を実装してみましょう。フォームの入力に不足があったとき、エラーの表示をして、データを再入力するように指示をする処理を実装してみます。このとき、前に入力した文字列があったらフォームに再表示するようにします。たとえば、タイトルのみが入力されていてURLが入力されていなかった場合。エラーの表示とともに、直前に入力したタイトルだけがあらかじめ入力された状態でフォームを表示するわけです。そうすることによって、入力の二度手間を防ぎ、使いやすいWebアプリケーションを作ることができます。

図03　フォームの入力に不足があった場合



テンプレートファイルを作る

まずは、Webアプリケーションの出力として利用するHTMLを、独立したテンプレートファイルとして設置します。テンプレートファイルには、状況によって内容が変化する場所に特別な文字列を埋め込みます。このテンプレートをひな形として、Webアプリケーションの出力となるHTMLを組み立てます。

今回は、標準モジュールに組み込まれているTemplateクラスを使います。Templateクラスでは、**${title}**というような特別な記法を使って、動的に変化する文字列を埋め込むようになっています。今回作るWebアプリケーションでは、次の2つの要素が動的に変化します。

- **フォームに埋め込まれる文字列**
- **エラーメッセージ**

フォームに埋め込まれる文字列は、inputエレメントのvalueアトリビュートに埋め込みます。テンプレートのvalueアトリビュートに、置換用の文字列を埋め込んでテンプレートを作ることになります。エラーメッセージはHTMLの内部に文字列として表示します。pエレメントなどで囲んで、エラーを表示したい場所に置換用の文字列を埋め込むことになります。

以下がテンプレートとして利用するファイルの内容です。「bookmarkform.html」というファイル名で、cgi-binフォルダの下に設置してください。もちろん、ファイルの文字コードはUTF-8にしておきます。

HTMLに紛れて${〜}という置換用文字列が4つ設置されています。この部分が置き換わって、Webアプリケーションの出力用文字列が作られます。

List01　bookmarkform.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-
8">
```

```
</head>
<body>
<h1>簡易ブックマーク</h1>
<p>${message}</p>
<form method="post" action="templatebbs.py">
    タイトル : <input type="text" name="title" size="40"
                      value="${title}" /><br />
    URL    : <input type="text" name="url" size="40"
                     value="${url}" /><br />
    <input type="hidden" name="post" value="1" />
    <input type="submit" />
</form>

${bookmarks}

</body>
</html>
```

Webアプリケーションのロジックを作る

次に、実際の処理を行うWebアプリケーションのプログラムを作ります。今回のプログラムには、以下のような機能を持たせます。

- **データを登録するフォームを表示する**
- **フォームから正しいデータが送られた場合、データを登録する**
- **データに不足があった場合、フォームを再表示する**

データの登録には、データベース（SQLite3）を使います。プログラム中では、DBAPIを使ってSQLを発行し、データの登録を行います。

以下がWebアプリケーションの本体となるプログラムです。cgi-binフォルダに「templatebbs.py」というファイル名で設置してください。文字コードはUTF-8にしてください。

プログラムがスッキリとしていて、見通しが良くなっていることが分かると思います。P.87で作成したhttphandlerモジュールのRequestクラス、Responseクラスを使うことによって、Webアプリケーションで実行する典型的な処理が抽象化され、プログラムが簡潔に書けるようになりました。その上今回は、テンプレートエンジンを使うことで、HTMLがプログラムから一掃されました。入力が正しいかどうかを検証し、正しかったらデータを登録する、というプログラムで行っている処理の内容が、かなり分かりやすくなっているはずです。

List02 templatebbs.py

```
#!/usr/bin/env python
# coding: utf-8

import sqlite3
from string import Template
from os import path
from httphandler import Request, Response, get_htmltemplate
import cgitb; cgitb.enable()

con=sqlite3.connect('./bookmark.dat')
cur=con.cursor()
try:
    cur.execute("""CREATE TABLE bookmark (
                   title text, url text);""")
except:
    pass

req=Request()
f=req.form
value_dic={'message':'', 'title':'', 'url':'','bookmarks':''}

if f.has_key('post'):                                              ❶
    if not f.getvalue('title', '') or not f.getvalue('url', ''):   ❷
        value_dic['message']=u'タイトルとURLは必須項目です'           ❸
        value_dic['title']=unicode(f.getvalue(
                            'title', ''), 'utf-8', 'ignore')
        value_dic['url']=f.getvalue('url', '')
    else:
        cur.execute(
            """INSERT INTO bookmark(title, url) VALUES(?, ?)""",
            (f.getvalue('title', ''), f.getvalue('url', '')))
        con.commit()

res=Response()                                                     ❹
f=open(path.join(path.dirname(__file__), 'bookmarkform.html'))
t=Template(unicode(f.read(), 'utf-8', 'ignore'))
```

```
body=t.substitute(value_dic)      ❺
res.set_body(body)
print res
```

さて、プログラムの内容について簡単に見てみましょう。まず、プログラムの冒頭では必要なモジュールをインポートしています。その後、try〜exceptを使ってデータベースにデータを記録するためのテーブルを作っています。

その後は、フォームから送られたデータの処理が続いています。フォームにはWebブラウザ上には見えないように**post**というキーが埋め込まれています。このため、フォームからデータが送られた場合には、必ず「post」というキーが送られてくるようになっています。このキーがあるかどうかを判別して、ブックマークを登録するかどうかを判断しています（❶）。

フォームからデータが送られた場合は、すべての項目が入力されているかどうかを調べます。もし入力に不足がない場合には、データベースにデータを登録します（❷）。

入力項目に不足がある場合は、エラー表示をしながら直前に入力された項目をフォームに表示するようなデータを作ります（❸）。エラー、フォームの入力として埋め込む文字列は、value_dicという辞書に登録します。辞書のキーとなるのは、テンプレートに埋め込んである置換用の文字列に出てきた文字列です。

最後に、出力の準備をします。まずはResponseクラスのインスタンスを作ります。その後、ファイルからテンプレートを読み、Templateクラスのインスタンスを作ります（❹）。その後、Templateクラスのインスタンスのsubstitute()メソッドを使ってテンプレートの変換を行います（❺）。このとき、テンプレートに埋め込みたいデータを辞書の形式で渡します。

今回のプログラムでは、表示をコントロールする部分と、処理を行う部分が完全に分離しています。表示上の修正を行いたい場合はテンプレートを直します。処理の内容を直したい場合にはプログラムを直します。このように、テンプレートエンジンを使うと、目的によって修正部分の切り分けがしやすくなるのです。

Webアプリケーションの場合、出力となるHTMLの修正を頻繁に行います。Webブラウザに HTMLを表示しながら、文字の大きさや位置、表示方法などを修正して、より分かりやすい出力を作る、というようなことをよく行います。そのような場合、出力のひな形がテンプレートファイルとして独立していた方が作業がはかどります。テンプレートファイルを直すだけで、文字の大きさやデザインなどの変更を行うことができるわけです。

ブックマークリストを埋め込む

ところで、このWebアプリには足りない機能があります。ブックマークは登録できるものの、登録したブックマークを表示することができません。このままではとても不便なので、登録済みのブックマークを埋め込む機能を追加しましょう。登録済みのブックマークは、フォームの下に表示するようにします。

図04　スクリプト変更後の画面

テンプレートの最後の方に**${bookmarks}**という置換用の文字列があります。この文字列はまだ使われていませんので利用することにしましょう。

Templateクラスを使ったテンプレートエンジンで実現できるのは、文字列の置換だけです。つまり、テンプレート上に置き換えて表示したい文字列は、プログラム側で作り、テンプレートエンジンに渡してやる必要があるわけです。

登録済みのブックマークはデータベースに入っています。DBAPIを使ってSQLite3にSQL（SELECT文）を送信すれば、登録済みのブックマーク一覧を取得できます。ブックマーク一覧を取得したら、それを元に表示用の文字列を組み立てます。

以下が、登録済みブックマークの一覧を表示するためのコードです。先ほどのtemplatebbs.pyに以下のコードを追加してください。「res=Response()」としてレスポンスオブジェクトを作っている行の直前に追加します。

List03 templatebbs.pyへの追加部分

```
                          ⋮
listbody=''
cur.execute("SELECT title, url FROM bookmark")
for item in cur.fetchall():
    listbody+="""<dt>%s</dt><dd>%s</dd>¥n"""%(item)
listbody="""<ul>¥n%s</ul>"""%listbody
value_dic['bookmarks']=listbody
                          ⋮
```

せっかくPythonのコードからHTMLを完全に追い出したのに、またHTMLが入り込んでしまいました。このようになってしまったそもそもの問題は、Templateクラスが単純な置換を行う機能しか持っていない、ということにあります。もっと複雑なWebアプリケーションを作るには、より高度な機能を持ったテンプレートエンジンがあった方が便利そうです。

テンプレートエンジンに求められる機能と分類

簡単なテンプレートエンジンを使ってWebアプリケーションを開発してみることで、テンプレートエンジンの有用性を確かめることができました。文字列の置換機能だけでも、ある程度のことは可能ですが、より高度な処理を行おうとすると、プログラムの中でHTML文字列を組み立てる必要が出てきます。

普通の文章のように一次元的な構造を持つテキストと異なり、HTMLは「構造」を持っています。HTMLでは、繰り返しによって要素を表現することがよくあります。今回追加したブックマークリストなどがその例です。また、Webアプリケーションのように比較的高度な出力を求められる場面では、**条件分岐**のような機能があると便利でしょう。

このように、Webアプリケーションで利用するテンプレートエンジンには、プログラミング言語に似た高度な機能が求められることがあります。

Pythonには、Webアプリケーション作成に活用できる既成のテンプレートエンジンが多くあります。実際、そのようなテンプレートエンジンでは、文字列置換以上の高度な機能を実装しています。置換用の文字列を埋め込む場合と同じように、特別なパターンをテンプレートに埋め込み、そのような機能を利用するようになっています。

以下では、典型的なテンプレートエンジンに実装されている主な機能について解説します。そうすることで、テンプレートエンジンに求められる機能について探ってみましょう。

ループ

既存のテンプレートエンジンはほとんど、テンプレート上のある部分を繰り返し表示する機能を持っています。Webアプリケーションの出力となるHTMLでは、似たような内容を繰り返して利用することがよくあります。複数の要素をリスト表示したい場面もあるでしょう。また、UIとなるフォームでは、メニューやラジオボタンなど、同じようなタグを複数回繰り返して表示する場面が多くあります。

表示する要素の内容が状況によって変わる場合は、なんらかのプログラミング的な方法を使って取り扱った方が効率がよくなります。このような処理をするために、テンプレートエンジンの持つループ機能を利用すると便利です。テンプレートエンジンを利用するプログラムの側では、ループの元となるシーケンスやイテレータを作ります。ループやイテレータを渡し、テンプレートエンジン側で繰り返し処理をします。

似たような要素をループを使って記述することで、テンプレートの記述自体がシンプルになる、という利点もあります。

条件分岐

Webアプリケーションでは、条件によって表示の内容や方法を切り替えたいことがよくあります。たとえば、エラーがある場合にのみエラーを表示する、ある値を超えた数値は目立つように表示する、というようなことをしたい場合に、Pythonのif文のような条件分岐を使えると便利です。実際、多くのテンプレートエンジンではこの条件分岐の機能が用意されています。

プログラム側では、条件の判断に必要なデータだけを作ってテンプレートに渡すことになります。テンプレート側には条件が記述してあって、場合に

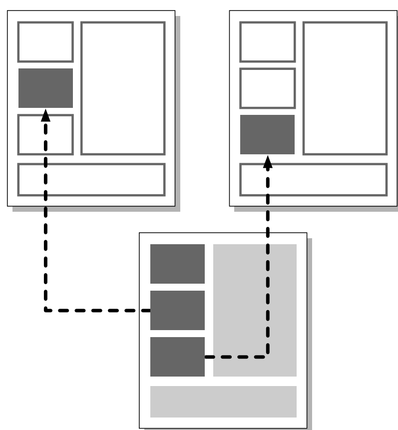
よって切り分けをして表示を切り替えます。

テンプレートの部品化

たとえばブログのサイドバーと呼ばれる部分のように、Webアプリケーションでは複数の場所で共通して利用される**部品**を活用することがあります。このように、共通して利用する部品を複数のテンプレートにバラバラに記述しておくのでは、開発の効率が落ちてしまいます。ある部品を修正したいときには、複数に散らばっているすべての部品を修正しなければならないからです。

Webアプリケーションでこのように共通した部分を実装するときに、テンプレートの一部を部品化できると便利です。既存のテンプレートエンジンでも、比較的高機能なものにはこの機能が実装されています。また、部品を持っている外部のテンプレートを呼び出すために、多くのテンプレートエンジンでは外部のテンプレートを読み込む機能を備えています。

図05　一部を部品化して他のテンプレートで利用できる機能を持つテンプレート



テンプレートの部品化は、Pythonのプログラミングでモジュールを作るのに似ています。Pythonでは、よく行う処理を関数としてまとめておき、いろいろなプログラムから再利用できるようにモジュールを作ります。テンプレートの部品化は、目的や手法がモジュール作成によく似ています。

マスターテンプレートとスロット

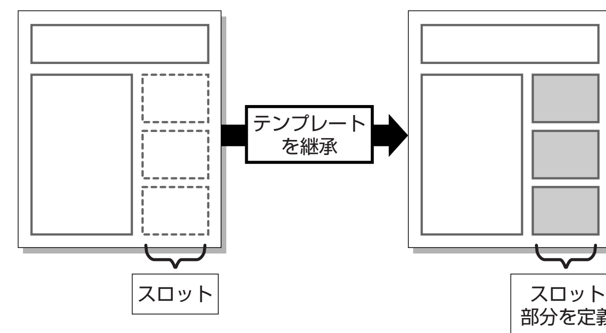
より高機能なテンプレートエンジンには、Webアプリケーションの出力となるHTMLの**配置**を決める機能が備わっていることがあります。最近のWebアプリケーションでは、画面のレイアウトをブロックに分割して配置を行います。ブロックの中には、場合によってはいつも決まった内容を表示することもありますし、場合によって表示内容が変化することもあります。

たとえば、ブログの画面を思い出してください。一般的なブログの画面には、ヘッダ、本文、サイドバーという3つのブロックがあります。このうち、ヘッダの表示内容はいつも変わりません。本文の表示内容は状況によって変わります。トップページであれば、近々のエントリ数個を表示します。エントリ単体の表示であれば、該当エントリの本文だけを表示します。また、サイドバーの表示も状況によって変わります。

このようにブロックに分割したレイアウトを効率よく扱うために、特殊な機能を備えているテンプレートエンジンがあります。全体的なレイアウトを定義したテンプレート（マスターテンプレート）を用意しておきます。実際に表示を行うテンプレートでは、変化するブロックだけを継承して定義します。このようにすることで、ブロックレイアウトで変化する部分を効率よく扱うことができます。

このように、テンプレートの全体のうち、変化する部分だけを定義して継承する機能のことを**スロット**と呼ぶことがあります。

図06　スロット



マスターテンプレートの機能は、Pythonの**抽象クラス**によく似ています。抽象クラスには、クラスの振る舞いのうち大まかな部分や、最小限度必要な部分のみを定義します。より専門的な処理を行うメソッドなどは、抽象クラスを継承したクラスに実装します。抽象クラスを継承したクラスに実装したメソッドが、テンプレートエンジンのスロットに相当する機能になります。

コードブロックの埋め込み

テンプレートエンジンによっては、テンプレート内にPythonのコードそのものを埋め込むことができるものがあります。実際、テンプレートの中で簡単な処理が実行できると便利な場面があります。テンプレートの中だけで利用する変数を定義したり、前段階で簡単な文字列処理をして、結果を表示に利用する、といった目的でコードを書くことが多いでしょうか。

テンプレートエンジンで処理をするテンプレートには、特殊な記法を使ってPythonのコードを埋め込みます。文字列の置換、ループや条件分岐などでは、Pythonの式を埋め込みます。Pythonの式とは、簡単に解説すると「改行を使わないで書けるコード」のことです。変数、関数呼び出しなどが式に相当します。

コードブロックでは、Pythonの**文**が記述できます。Pythonの文を簡単に解説すると「改行を組み合わせて記述するコード」のことです。変数の定義、for文を使ったループ、if文を使った条件分岐などが文に相当します。

テンプレートの中にPythonのプログラムが書けることが分かると、なんでもテンプレートの内部で実行したくなるかもしれません。しかし、あまり複雑な処理をテンプレート内で実行しようとすると、Webアプリケーションのプログラムとの役割分担が曖昧になってしまいます。プログラムの内部からHTMLの文字列を追い出すことでプログラムがスッキリしました。テンプレートの内部にPythonのコードが氾濫するようになると、今度はテンプレートの見通しが悪くなってしまいます。

テンプレートエンジンの中には、文を使うような複雑なコードはテンプレート内部に書くべきではない、と主張してこの機能を実装していないものもあります。

テンプレートエンジンの分類

Pythonには、Webアプリケーション開発に利用できるテンプレートエンジンが数多くあります。それぞれが持っている機能はとてもよく似ています。また、特殊なパターンや記法を使って、動的に生成した文字列を埋め込むという基本的な考え方も同じです。

ただし、テンプレートエンジンによって記法として採用しているパターンが異なります。また、パターンの埋め込み方には大まかに分けて2つの手法があります。

1つは、場所を問わず、テンプレート専用の記法でパターンを埋め込むタイプです。ここでは埋め込み型と呼びましょう。HTMLのような文字列に、特定のパターンを持った文字列を埋め込みます。

以下は、**埋め込み型**のテンプレートであるDjangoの記法のサンプルです。liエレメントの要素を繰り返し、複数表示しています。{%〜%}や{{〜}}という文字列で囲み、処理の内容を指示していることが分かります。独自の記法で囲まれた部分が、出力時に動的に置き換わります。

Djangoを使用したコード

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

もう1つのタイプは、HTMLのタグ（エレメント）の内部にテンプレート独自のパターンを埋め込むタイプです。こちらのタイプを**XML型**と呼びましょう。

エレメントの中にテンプレート用の文字列を埋め込むため、でき上がったテンプレートをHTMLやXMLとして評価したときに**正しく**（valid）なります。テンプレートの仕様自体、端正な出力をするように意図されて作られているのです。半面、テンプレートの記法に制限が多くなります。

以下は、XML型テンプレートGenshiので書いたサンプルです。py:〜というアトリビュートの形で、テンプレート独自のパターンが埋め込まれているのが分かります。py:から始まるアトリビュートは、出力時には消されます。

なお、py:contentというアトリビュートは特殊な働きをします。このアトリビュートに記載されている内容が、エレメントの内部（この場合は<li>タグで囲まれた部分）に埋め込まれるのです。

Genshiを使用したコード

```
<ul>
  <li py:for="choice in choices"
      py:content="choice">
    A choice here
  </li>
</ul>
```

09-04　Pythonでテンプレートエンジンを作る

標準モジュールに内蔵されているTemplateクラスだけでは、高度なWebアプリケーションを作るには機能不足である、ということはよく分かっていただけたと思います。既存のテンプレートエンジンは、単純な文字列置換だけでなく、より高度な機能を装備しています。

ここでは、少し趣向を変えて、先ほど使ったTemplateクラスよりも高機能なテンプレートエンジンを作ってみましょう。ただし、あまり機能は欲張らず、**置換機能**と**条件分岐**、**ループ**の機能だけを実装することにします。シンプルな機能を持ったテンプレートエンジンを作ることによって、テンプレートエンジン自体への理解を深めることもできるはずです。

このテンプレートエンジンを「SimpleTemplate」と呼ぶことにします。

テンプレートエンジンの仕様を決める

テンプレートエンジンを作るに当たって、まずは内部に埋め込む命令の記法などを決めなければなりません。

テンプレートエンジンのタイプとしては埋め込み型を選ぶことにします。埋め込み型の方が比較的実装が簡単だからです。HTMLに埋め込み、動的に置き換える文字列や命令を指定する部分は、目立っている必要があります。その方が、HTMLの他の部分と見分けやすく、修正をするときに目的の場所をすぐに見つけられるはずです。

SimpleTemplateでは、**$**という記号に特別な意味を持たせるようにしましょう。動的に置き換える要素は**${〜}**で囲むようにします。この中には、テンプレートエンジンに渡された辞書のキーや、Pythonの式を書けるようにします。

条件分岐やループは、行頭から始まって**$if〜:**、**$for〜:**と記述することにします。条件式、ループ変数やシーケンスの書き方はPythonに準じることにします。条件分岐では**$else:**や**$elif〜:**は利用できません。

また、条件分岐やループのブロックの「終わり」を示すために、**$endif**と**$endfor**というキーワードを使います。このように明示的に終わりを指定しないと、どこからどこまでがループや条件分岐で処理すべき範囲なのかが分からなくなってしまいます。こうしてみると、インデントを使ってブロックの範囲を指定するPythonの記法がいかにシンプルかがよく分かります。

実装の指針

テンプレートエンジンはクラスとして実装します。テンプレート本文となる文字列、またはテンプレートファイルのパスを渡してクラスのインスタンスオブジェクトを生成します。

テンプレートのクラスには、テンプレートを解釈して結果を出力するメソッドを作ります。本来、このような処理を実装するためには**トークナイザー**と呼ばれる仕組みを組み込む必要があります。今回作るテンプレートエンジンの仕様では、範囲の判別が必要な要素が必ず行頭から始まっています。このような単純な仕様のため、テンプレート本文を行ごとに分割して処理を進めることができます。なお、テンプレートエンジン内部では、文字列をユニコード文字列として扱うようにします。

${〜}で囲まれた部分は、見つけたその場で置き換えを実行します。条件分岐、ループについては、まず処理対象となる範囲を探し出し、その範囲のみを対象に処理をする必要があります。この部分が今回の実装のキモとなります。

処理の内容は、分かりやすく、とても素朴に書いてあります。処理速度や機能を優先するなら、別の実装方法があるはずですが、処理の内容が分かりやすいように、あえて簡単なプログラムにしてあります。またモジュール名はsimpletemplateとします。

初期化部分の実装

まずは、クラスの初期化部分を実装してみましょう。クラスの宣言と、初期化用の__init__()メソッドを実装します。必要であれば本文をファイルから取得、改行で分割し、インスタンスのbodyというアトリビュートに保存します。

≡List04 __init__メソッドの実装(simpletemplate.py)

```
#!/usr/bin/env python
# coding: utf-8
import re

if_pat=re.compile(r"¥$if¥s+(.*¥:)")                              ❷
endif_pat=re.compile(r"¥$endif")
for_pat=re.compile(r"¥$for¥s+(.*)¥s+in¥s+(.*¥:)")
endfor_pat=re.compile(r"¥$endfor")
value_pat=re.compile(r"¥${(.+?)}")

class SimpleTemplate(object):
    """
    シンプルな機能を持つテンプレートエンジン
    """
    def __init__(self, body='', file_path=None):
        """
        初期化メソッド
        """
        if file_path:
            f=open(file_path)
            body=unicode(f.read(), 'utf-8', 'ignore')
        body=body.replace('¥r¥n', '¥n')
        self.lines = body.split('¥n')                             ❶
        self.sentences = ((if_pat, self.handle_if),
                          (for_pat, self.handle_for),
                          (value_pat, self.handle_value),)
```

今回のテンプレートエンジンでは、動的置き換えに使うパターンをタプルとしてアトリビュートに保存しておきます。パターンの判別を行う正規表現オブジェクトと、パターンにマッチしたときに呼び出されるメソッドの呼び出し可能オブジェクトをタプルにし、タプルとして並べておきます(❶)。

そして正規表現オブジェクトの定義部分が❷です。

レンダリング処理を実装する

では次に、テンプレートに埋め込まれたパターンを展開して、テンプレートをレンダリングする処理について解説します。以下が該当部分のコードです。クラスメソッドで、レンダリングの処理を行っています。

≡List05 process()メソッドの実装(simpletemplate.py)

```
    def process(self, exit_pats=(), start_line=0, kws={}):
        """
        テンプレートのレンダリング処理をする
        """
        output=u''
        cur_line=start_line
        while len(self.lines) > cur_line:
            line=self.lines[cur_line]
            for exit_pat in exit_pats:
                if exit_pat.search(line):
                    return cur_line+1, output
            for pat, handler in self.sentences:
                m=pat.search(line)
                pattern_found=False
                if m:
                    try:
                        cur_line, out=handler(m, cur_line, kws)
                        pattern_found=True
                        output+=out
                        break
                    except Exception, e:
                        raise
                         "Following error occured in line %d¥n%s" ¥
                         %(cur_line, str(e))
            if not pattern_found:
                output+=line+'¥n'
            cur_line+=1
        if exit_pats:
            raise "End of lines while parsing"
        return cur_line, output
```

メソッドでは、インスタンスに保存されているテンプレート本文を1行ずつ読み込みながら処理を進めています。テンプレート本文を最後まで読み込んだら処理を終える、という処理を実現するため、while文を使ってループ

を組んでいます。メソッドの引数として**終了条件**を渡せるようになっています。テンプレート本文を1行ずつ読んでいき、この終了条件に見合う行が出現した場合も処理を終了します。終了条件は正規表現のパターンとして与え、シーケンスに複数指定できるようになっています。

ループの中では、初期化メソッドで定義した置き換えパターンを使い、テンプレートの各行を評価しています。while文のループの中にfor文のループが見えますが、この部分が処理を実行している場所です。もし、置き換えパターンに設定された正規表現にヒットする行が現れたら、パターンを処理するためのメソッドを呼び出します。for文の繰り返し変数としてパターンの処理メソッドの呼び出し可能オブジェクトを受け取っていますので、この変数に対して呼び出しを行っています。変数にメソッドが入っているというのは一見奇妙に見えるかもしれませんが、Pythonではよく使われる手法です。

パターンの処理メソッドでは、置換やループ、条件分岐などの処理が実行されます。ループや条件分岐のように**範囲**に対して実行する処理では、process()メソッド自体を再帰呼び出しして利用します。このようにすることで、入れ子になったループや条件分岐、条件分岐の中にあるループなど、複雑な構造を持ったテンプレートを処理できるようになります。

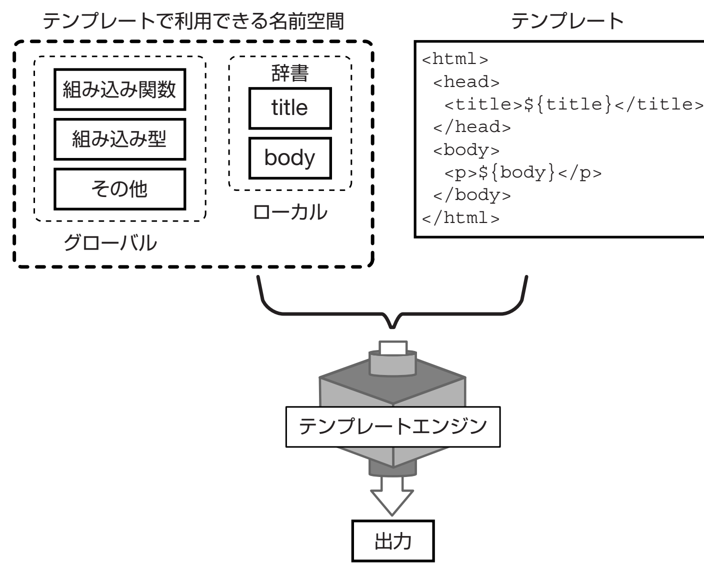
パターンの置換を処理する

SimplaTemplateでは、${〜}で囲まれた部分を置換用の文字列として扱います。パターンで囲まれた部分はPythonの式と見なし、式の返す値を置換してテンプレートに埋め込みレンダリングします。変数名が埋め込まれていれば変数の内容を文字列に変換して埋め込みます。関数呼び出しであれば、関数の戻り値を文字列として埋め込みます。

テンプレートエンジンは、プログラムから埋め込みに利用する変数などのオブジェクトを受け取ってレンダリングの処理をします。標準モジュールのTemplateクラスでは、辞書としてレンダリングに利用する変数などを渡していました。辞書のキーを変数名に、値を変数に代入されたオブジェクトのように扱い、テンプレートをレンダリングしているわけです。引数として渡された辞書を使って、テンプレート内部で利用する**名前空間**を作っているわけです。辞書で渡されたオブジェクトは、Pythonのプログラムで言う**変数**などが定義される名前空間になります。そのような名前空間は**ローカルの名前空間**と呼ばれます。その他にも、組み込み関数などが置かれる名前空間も利用できます。

このメソッドでは、**_kws**という引数がその辞書に該当します。この辞書には、Webアプリケーションのプログラムから渡された辞書が渡ってきます。この辞書に入ったデータをローカル変数のように見立て、テンプレートのレンダリングを行います。辞書にはいろいろなPythonのオブジェクトを代入できます。変数はもちろんPythonのオブジェクトですし、インスタンス、関数やメソッド、モジュールもPythonのオブジェクトです。このようなオブジェクトをテンプレートエンジンに渡せば、いろいろな処理が実行できることになります。

図07　名前空間



List06　handle_value()メソッドの実装（simpletemplate.py）

```
    def handle_value(self, _match, _line_no, _kws={}):
        """
        ${...}を処理する
        """
```

```
        _line=self.lines[_line_no]                                    ❶
        _rep=[]
        locals().update(_kws)                                         ❷
        pos=0
        while True:
            _m=value_pat.search(_line[pos:])
            if not _m:
                break
            pos+=_m.end()
            _rep.append( (_m.group(1), unicode(eval(_m.group(1)))) )  ❸
        for t, r in _rep:
            _line=_line.replace('${%s}'%t, r)
        return _line_no, _line+'¥n'
```

テンプレート内でパターンの置換を行っているメソッドについて、処理内容を詳しく解説しましょう。メソッドには、正規表現のマッチオブジェクト、行数、テンプレートに渡された辞書が引数として渡ってきます。

まずは、引数として渡ってくる行数を使い、処理対象となるテンプレート本文の行を取り出します（❶）。

その後に、❷のような奇妙な行が見えます。この行がこのメソッドの第1のキモです。locals()はPythonの組み込み関数で、ローカル変数を定義している辞書を返します。この辞書に対してupdate()メソッドを呼び出しています。引数として、テンプレート内で変数として利用するオブジェクトが入った辞書を渡します。このようにすると、辞書を元にローカル変数を定義できるのです。インタラクティブシェルを使って簡単な例を試してみましょう。aという変数は明示的に定義していませんが、locals()の返す辞書に対して操作を行うことで変数定義と同様の処理が実現できていることが分かります。

```
>>> print a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> locals().update({'a':1})
>>> print a
1
```

このようにして、テンプレート内で利用するローカル変数を定義します。その後は、処理対象となる行から置換用のパターンを探し出します。パターンが見つかったら、内部の文字列をPythonの式と見なし、返ってきた結果を文字列として埋め込む処理をします（❸）。文字列をPythonの式と見なす処理にはeval()を使っています。eval()は文字列を引数としてとり、文字列をPyhtonの式として評価して結果を返す関数です。

このようにして、行にある置換用のパターンを変換していきます。結果を返して処理を終了し、process()メソッドに戻ります。

条件分岐の処理

条件分岐のパターンは、ある条件によって表示内容を切り替えたいときに利用します。SimpleTemplateでは、**$if〜:**というパターンを使って条件分岐を埋め込みます。条件分岐の際に評価するブロックの終端を表現するために**$entif**というパターンを使います。**$if〜:**と**$endif**で囲まれた部分が、条件分岐で評価される範囲となります。

$ifの後には、Pythonの式を記述します。この式を真偽値として評価して、Trueに相当するかFalseに相当するかによって、処理の内容を振り分けます。

List07 handle_if()メソッドの実装（simpletemplate.py）

```
    def handle_if(self, _match, _line_no, _kws={}):
        """
        $ifを処理する
        """
        _cond=_match.group(1)
        if not _cond:
            raise "SyntaxError: invalid syntax in line %d" % line_no ❶
        _cond=_cond[:-1]
        locals().update(_kws)
        _line, _out=self.process((endif_pat, ), _line_no+1, _kws)
        if not eval(_cond):                                          ❷
            _out=''
        return _line-1, _out
```

条件分岐を行っている部分について解説しましょう。条件分岐の処理をする前に、表記のエラーをチェックしています。**$if**の後に条件となる式があるかどうかを簡易にチェックし、もしない場合はエラーを返しています（❶）。

もし条件式が見つかったら、該当部分を文字列として取り出します。

条件分岐を行うときには、Webアプリケーションのプログラムから渡された変数や、インスタンスなどのオブジェクトを利用したいことがあるはずです。このため、パターン置換のときと同じ方法で引数の辞書の内容をローカル変数として展開します。その後、条件式の文字列をeval()で評価、if文を使って結果によって処理を振り分けています（❷）。

条件式が真だった場合には、ブロックの**$endif**までの部分をレンダリングします。レンダリングするブロックの中に、条件分岐やループが入れ子になっている場合を考慮して、ブロック内のみを対象として処理するようにprocess()メソッドを再帰的に呼び出しています。

条件式が偽だった場合には、ブロック内をレンダリングしません。

ループの処理

同じ要素をリスト風に表示したり、UIとなるフォームのメニューを動的に作るときに利用するのがループの機能です。条件分岐と同様に処理をする範囲を明示する必要があります。SimpleTemplateでは**$for～in～:**というパターンと**$endfor**というパターンで囲まれる範囲を処理の対象とします。ループの処理は、Pythonの文法と同じく、シーケンスやイテレータを元に行います。繰り返し変数にシーケンスやイテレータの要素を1つずつ代入していき、ブロックの内容をレンダリングします。要素がなくなったらループの処理を終了します。

List08 handle_for()メソッドの実装（simpletemplate.py）

```
    def handle_for(self, _match, _line_no, _kws={}):
        """
        $forを処理する
        """
        _var=_match.group(1)                                    ❶
        _exp=_match.group(2)
        if not _var or not _exp:
            raise "SyntaxError: invalid syntax in line %d" % line_no
        locals().update(_kws)                                   ❷
        _seq=eval(_exp[:-1])                                    ❸
        _out=''
        if not _seq:
            return self.find_matchline(endfor_pat, _line_no), _out
        for _v in _seq:                                         ❹
            _kws.update({_var:_v})
            _line, _single_out=self.process((endfor_pat, ),
                 _line_no+1, _kws)
            _out+=_single_out
        return _line-1, _out
```

メソッド内では、まず**$for～:**以下のパターンが正しいかどうかを簡単にチェックしています（❶）。ループを実行するためには、繰り返し変数の変数名と、シーケンスやイテレータに相当するPythonの式が必要です。この2つのパターンが存在しない場合は、エラーとして扱っています。

ループを行うときに利用するPythonの式には、Webアプリケーションのプログラムから渡されたオブジェクトを使うことが多いはずです。そのためここでも、引数として渡された辞書をローカル変数として登録しています(❷)。その後、繰り返しを実行するための式に相当する文字列を、eval()を使ってPythonのオブジェクトに変換しています（❸）。

eval()で変換した式を元に、for文を使ってループを組んでいます。ループの中では、local()を返す辞書をupdate()し、繰り返し変数を定義しています。その上で、**$for～:～$endfor**内のループブロックの範囲をprocess()メソッドに渡して処理をしています（❹）。ループブロックの中に、置換用のパターンや条件分岐、他のループがある場合に対応できるよう再帰呼び出しを行っているのです。

そのほかの処理

上記で解説した以外の処理としては、実際にテンプレートの表示を行うためのrender()メソッドと、正規表現オブジェクトを受け取りマッチする行の行数を返すfind_matchline()メソッドがありますが、ここでは詳しく解説しません。簡単なので自分で研究してみてください。

完成したsimpletemplate.pyは、次のとおりです。

List09 simpletemplate.py（完成版）

```
#!/usr/bin/env python
# coding: utf-8
import re

if_pat=re.compile(r"¥$if¥s+(.*¥:)")
endif_pat=re.compile(r"¥$endif")
for_pat=re.compile(r"¥$for¥s+(.*)¥s+in¥s+(.*¥:)")
endfor_pat=re.compile(r"¥$endfor")
value_pat=re.compile(r"¥${(.+?)}")

class SimpleTemplate(object):
    """
    シンプルな機能を持つテンプレートエンジン
    """
    def __init__(self, body='', file_path=None):
        """
        初期化メソッド
        """
        if file_path:
            f=open(file_path)
            body=unicode(f.read(), 'utf-8', 'ignore')
        body=body.replace('¥r¥n', '¥n')
        self.lines = body.split('¥n')
        self.sentences = ((if_pat, self.handle_if),
                          (for_pat, self.handle_for),
                          (value_pat, self.handle_value),)

    def render(self, kws={}):
        """
        テンプレートをレンダリングする
        """
        l, o=self.process(kws=kws)
        return o

    def find_matchline(self, pat, start_line=0):
        """
        正規表現を受け取り，マッチする行の行数を返す
        """
        cur_line=start_line
        for line in self.lines[start_line:]:
            if pat.search(line):
                return cur_line
            cur_line+=1
        return -1

    def process(self, exit_pats=(), start_line=0, kws={}):
        """
        テンプレートのレンダリング処理をする
        """
        output=u''
        cur_line=start_line
        while len(self.lines) > cur_line:
            line=self.lines[cur_line]
            for exit_pat in exit_pats:
                if exit_pat.search(line):
                    return cur_line+1, output
            for pat, handler in self.sentences:
                m=pat.search(line)
                pattern_found=False
                if m:
                    try:
                        cur_line, out=handler(m, cur_line, kws)
                        pattern_found=True
                        output+=out
                        break
                    except Exception, e:
                        raise
                         "Following error occured in line %d¥n%s" ¥
                          %(cur_line, str(e))
            if not pattern_found:
                output+=line+'¥n'
                cur_line+=1
        if exit_pats:
            raise "End of lines while parsing"
        return cur_line, output

    def handle_value(self, _match, _line_no, _kws={}):
        """
        ${...}を処理する
        """
        _line=self.lines[_line_no]
```

```
        _rep=[]
        locals().update(_kws)
        pos=0
        while True:
            _m=value_pat.search(_line[pos:])
            if not _m:
                break
            pos+=_m.end()
            _rep.append( (_m.group(1), unicode(eval(_m.group(1)))) )
        for t, r in _rep:
            _line=_line.replace('${%s}'%t, r)
        return _line_no, _line+'¥n'

    def handle_if(self, _match, _line_no, _kws={}):
        """
        $ifを処理する
        """
        _cond=_match.group(1)
        if not _cond:
            raise "SyntaxError: invalid syntax in line %d" % line_no
        _cond=_cond[:-1]
        locals().update(_kws)
        _line, _out=self.process((endif_pat, ), _line_no+1, _kws)
        if not eval(_cond):
            _out=''
        return _line-1, _out

    def handle_for(self, _match, _line_no, _kws={}):
        """
        $forを処理する
        """
        _var=_match.group(1)
        _exp=_match.group(2)
        if not _var or not _exp:
            raise "SyntaxError: invalid syntax in line %d" % line_no
        locals().update(_kws)
        _seq=eval(_exp[:-1])
        _out=''
        if not _seq:
            return self.find_matchline(endfor_pat, _line_no), _out
        for _v in _seq:
            _kws.update({_var:_v})
            _line, _single_out=self.process(
                (endfor_pat, ), _line_no+1, _kws)
            _out+=_single_out
        return _line-1, _out
```

ブックマーク管理Webアプリを書き換える

さて、今回作ったシンプルなテンプレートエンジンを使って、先ほど作ったブックマーク管理Webアプリを書き換えてみましょう。SimpleTemplateは、テンプレート内に繰り返しを行う機能を持っています。この機能を使うと、HTMLに相当する文字列を完全にプログラムから駆除できるはずです。

まず、Webアプリケーションの出力となるテンプレートを書きます。SimpleTemplateは、埋め込みのパターンが標準ライブラリのTemplateクラスと同じです。そのため、フォームのvalueアトリビュートの部分は書き換える必要がありません。エラーメッセージを表示している部分で条件分岐を使い、既存ブックマークを表示している部分でループの機能を利用することにしましょう。

以下がSimpleTemplate用に書き換えたテンプレートです。「stbookmarkform.html」というファイル名でcgi-binフォルダに設置します。

List10 stbookmarkform.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
</head>
<body>
<h1>簡易ブックマーク</h1>

$if message:
<p>${message}</p>
$endif

<form method="post" action="">
    タイトル : <input type="text" name="title" size="40"
                     value="${title}" /><br />
```

```
    URL    : <input type="text" name="url" size="40"
                    value="${url}" /><br />
    <input type="hidden" name="post" value="1" />
    <input type="submit" />
</form>
<ul>

$for item in bookmarks:
    <dt>${item[0]} </dt>
    <dd>${item[1]} </dd>
$endfor

</ul>
</body>
</html>
```

次に、Webアプリケーションの処理を行うプログラムを書き換えます。フォームから送られたデータを元に、新しいブックマークを登録する部分は以前のプログラムと共通して利用できます。変更する必要があるのは、ブックマーク一覧に相当する文字列を作っている部分と、テンプレートエンジンを使ってWebアプリケーションの出力を作っている部分のみです（網掛けの部分）。HTMLが完全になくなり、プログラムがスッキリして見通しがよくなっているのが分かるはずです。

List11 stemplatebbs.py

```
#!/usr/bin/env python
# coding: utf-8

import sqlite3
from string import Template
from os import path
from httphandler import Request, Response, get_htmltemplate
from simpletemplate import SimpleTemplate

con=sqlite3.connect('./bookmark.dat')
cur=con.cursor()
try:
    cur.execute("""CREATE TABLE bookmark (
                    title text, url text);""")
```



```
except:
    pass

req=Request()
f=req.form
value_dic={'message':'', 'title':'', 'url':'','bookmarks':''}

if f.has_key('post'):
    if not f.getvalue('title', '') or not f.getvalue('url', ''):
        value_dic['message']=u'タイトルとURLは必須項目です'
        value_dic['title']=unicode(f.getvalue(
                    'title', ''), 'utf-8', 'ignore')
        value_dic['url']=f.getvalue('url', '')
    else:
        cur.execute(
            """INSERT INTO bookmark(title, url) VALUES(?, ?)""",
               (f.getvalue('title', ''), f.getvalue('url', '')))
        con.commit()

cur.execute("SELECT title, url FROM bookmark")            ❶
value_dic['bookmarks']=tuple(cur.fetchall())

res=Response()                                             ❷
p=path.join(path.dirname(__file__), 'stbookmarkform.html')
t=SimpleTemplate(file_path=p)
body=t.render(value_dic)
res.set_body(body)
print res
```

テンプレートエンジンでは、既存ブックマークをシーケンスとして受け取り、ループを使ってブックマークを表示しています。Webアプリケーションのプログラム側では、辞書にシーケンス（タプル）を渡すだけでよいわけです（❶）。

テンプレートで表示するデータができ上がったら、テンプレートのパスを指定してテンプレートエンジンのインスタンスオブジェクトを作ります。インスタンスのrender()メソッドを呼び出し、テンプレートをレンダリングします（❷）。後の手順はこれまでと同じです。

このように、高度な機能を持つテンプレートエンジンを使うと、プログラ

ムをスッキリ書けるようになります。表示をコントロールするための処理はテンプレート側に埋め込みます。プログラム側ではデータを操作したり表示に必要なデータを取り出す作業を担当します。

プログラムとテンプレートの役割分担を明確にして、処理を分担することで、Webアプリケーション全体の見通しがよくなるのです。見通しがよくなれば、開発がより効率的になります。開発が効率化すれば、より複雑なWebアプリケーションを簡単に作れるようになりますし、プログラムの拡張や修正も楽になります。最近では、より複雑で高機能なWebアプリケーションを作るために、高機能なテンプレートエンジンは必須となっています。

CHAPTER 10

O/Rマッパーを使ったデータベースの操作

テンプレートエンジンを使うことによって、Webアプリケーションのプログラムから HTMLを完全に追い出すことができました。結果として、プログラムがスッキリし、見通しがよくなりました。

そもそもプログラムとは、データを処理し、結果を返す、という手続きを記述するためにあるものです。Webアプリケーションであれば、データを受け取って加工したり、必要に応じてデータベースなどに保存する、または保存したデータをデータベースから取り出して処理をする、というようなことが主な処理の内容になります。

そのような処理の中に、HTMLのようなプログラムとは異質な文字列が紛れ込んでいると、結果としてプログラムの見通しが悪くなってしまいます。Webアプリケーションの出力となるHTMLを組み立てる処理はたいてい単調で冗長です。HTMLのような**異質**な文字列だけでなく、本質的でない処理がプログラムに紛れ込むことも、プログラムの見通しを悪くする原因となります。

HTMLと同様に、データベースとの通信を行うときに利用する**SQL**もまた、Webアプリケーションのプログラムに紛れ込む異質な文字列と言えるかもしれません。データベースの操作も、HTMLを文字列として組み立てる作業と似て、単調な文字列処理の繰り返しになりがちです。SQLをWebアプリケーションのプログラムから追い出すことができれば、プログラムはもう少しスッキリするはずです。

10-01　テーブルとクラスの関係

実際に、SQLをプログラムから追い出す方法について考える前に、データベースについてもう少し考察を深めてみましょう。

データベースでは、データを表のような形式にして保存します。表には、保存したいデータの種類の数だけカラムを作ります。どのような表を作るかを定義するのがテーブルです。データベースにデータを保存するためには、まずテーブルを作る必要があります。テーブルは、いわば保存するデータの**型**の定義に相当します。

データベースに保存するデータは、テーブルに定義されたデータの型に応じた形式で保存します。テーブルをひな形としてデータの型を決めて、データを保存していくわけです。

この**テーブル**と**データ**の関係は、ちょうどPythonのようなオブジェクト指向言語の**クラス**と**インスタンスオブジェクト**の関係に似ています。クラスには型を定義しますので、テーブルに似た性質を持っています。クラスインスタンスはクラスという型をひな形にして作りますので、データベース上のデータ（列）に似た性質を持っていると言えます。

テーブルとクラスの違い

テーブルとクラス、データとインスタンスオブジェクトは似た面がある半面、異なった性格も持っています。

たとえば、データベースのテーブルは単に**入れ物の形**を定義しているにすぎません。データベースから取り出したデータも単なる**値**です。入れ物からデータを取り出したり追加をするためには、**SQL**という特殊な文法を持った文字列を組み立てる必要があります。

データと、データを操作するための手続きが明確に分かれているのがデータベースの特徴といえます。

一方、Pythonのクラスは単なる入れ物ではありません。また、インスタンスオブジェクトもただのデータではありません。クラスもインスタンスも、定義されているデータに関する処理を実行することができます。Pythonの場合は、インスタンスオブジェクトやクラス自体を対象にメソッドという形式で処理を実行します。たとえばPythonのリスト型では、リストの反転、ソート順の変更など、データを処理するときに便利な**メソッド**がたくさん用意されています。

データと手続きが一体になっているのが、クラスやインスタンスオブジェクトの流儀です。

データベースとクラスの違いについて簡単に見たことろで、次にいくつかの典型的な処理を例にとってみましょう。データベースとクラスの間で、処理方法がどのように違うかを具体的に比較したいと思います。

データを登録する場合

データベースの場合は、INSERT文というSQLの文字列を作ってデータベースに送信することでデータを登録します。登録したいデータは、数値であっても文字列に変換してSQLに埋め込みます。SQLは以下のようになります。大文字で書いてあるのはSQLのうち定型の部分です。小文字で書いてある文字はカラム名を指定したり、データを指定する部分で、状況によって変化します。

```
INSERT INTO testtable (row1, row2) VALUES (1, 'foo')
```

Pythonのクラスの場合は、クラス自体を関数のように呼び出してインスタンスオブジェクトを作ります。登録したいデータは、引数の形で渡すことがほとんどです。数値、文字列など、登録したいオブジェクトをそのまま渡すことができます。

SQL文字列を組み立て、登録したいデータを文字列に変換する必要があるなど、データベースの操作の方が多少面倒なようです。ただし、どちらの処理も似通っていて、どちらが取り立てて大変、というわけではなさそうです。インスタンスを生成するコードは以下のようになります。

```
ins=TestClass(att1=1, att2='foo')
```

特定のデータを更新する場合

データベースの場合は、UPDATE文というSQL文字列を組み立てる必要があります。SQLには、更新したいデータのカラム名とデータそのものの他に、更新したいデータを特定するための情報を記載する必要があります。IDのような特別な情報を使って、更新する対象となるデータを指定する必要があるわけです。SQL文字列の書き方が面倒ですし、コードを見てもなにを実行しようとしているのか分かりづらくなってしまうかも知れません。

```
UPDATE testtable SET row1=2, row2='bar' WHERE id=1
```

Pythonクラスの場合は、インスタンスのアトリビュートにデータを保存することがほとんどです。アトリビュートのデータを更新するためには、イコールを使ってアトリビュートに代入を行うか、**アクセサ**と呼ばれるメソッドを呼び出します。データそのものに対して操作が行えるため、コードの書き方も簡単になりますし、処理の内容を直感的に把握しやすくなるはずです。

```
ins.row1=2
ins.row2='bar'
```

データの更新という点でみると、データベースよりクラスを使った処理の方が簡潔に実行できることが分かります。簡潔に実行できる処理は、コードも短くて済みますし、短いコードであれば処理の内容も把握しやすくなるはずです。

また、データベースの場合は、更新の処理をSQLという文字列に埋め込まなければなりません。そのため、Pythonのコードに比べて処理の実現方法が大きく異なっているのがよく分かります。

このように、データを扱う処理の一部では、クラスやインスタンスオブジェクトの方がより短く、分かりやすく処理を実行できるのです。

テーブルとクラス、データとインスタンスオブジェクトは互いによく似た関係にあります。もし、データベース上のクラスをクラスとして、データをインスタンスオブジェクトとして表現できたら、とても便利なはずです。

まず、SQLをPythonのプログラムから追い出すことができます。その上、データベースの処理を純粋なPythonのプログラムとして書けるようになります。SQLのようにPythonのプログラムと違った異質なものを追い出して、より純粋なPythonのコードだけをプログラムに書けるようになれば、より見通しがよいプログラムになるはずです。

O/Rマッパーとは

すでに解説してきたとおり、データベースでデータを扱う手法と、Pythonのようなオブジェクト指向的なデータの扱い方の間には大きなギャップがあります。データベースでは、データとデータを操作するための手続きが完全に分離しています。対してオブジェクト指向言語では、データと手続きが一体になっています。データを扱うときの考え方がそもそも異なるので、プログラムの中でデータベースを扱うときには、非Pythonな方法でデータを扱う必要が出てきます。

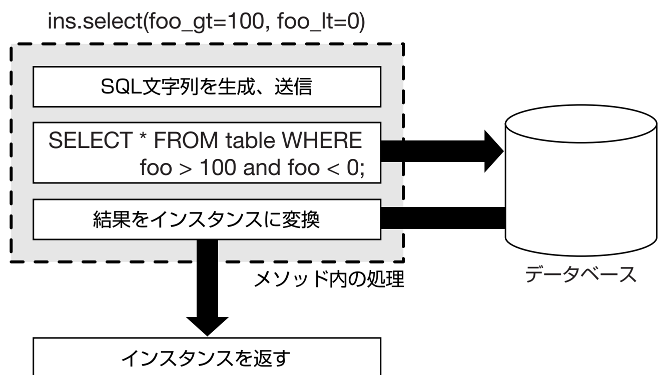
Webアプリケーションに限らず、データの操作を行う処理は、プログラムの基本部分といってよいくらい重要な部分です。そのような重要な部分に、非Python的な手法を使わなければならないとすると、プログラムは手軽に書けなくなってしまいます。インスタンスの生成、アトリビュートへの代入やメソッド呼び出しなど、Python的な手法を使ってデータベースを操作できれば、もっと手軽に、かつ簡潔にプログラムが書けるようになるはずです。

O/Rマッパーは、データベースとオブジェクト指向言語の間にあるギャップを埋める役割でよく利用される仕組みです。「O」は「オブジェクト」、「R」は「リレーショナル」を意味します。オブジェクト指向言語で利用されるオブジェクトと、リレーショナルデータベースのデータをうまくマッピングし、間を取り持ってくれる仕組みのことを指します。

O/Rマッパーにはたくさんの種類があり、マッピングの手法もいろいろとあります。O/Rマッパー全体に共通しているのは、**データベース上のデータをオブジェクトとして扱える**という特徴です。データを取り出したり、データを更新するために、SQL文字列を作る必要がほとんどありません。数値や文字列など、ごく普通のデータと同じように、データベース上のデータを扱えるのです。

O/Rマッパーを使っても、データベースと通信をするためにはどこかで誰かがSQL文字列を作る必要があります。データベースとの実際の通信は、O/Rマッパーが裏側で密かに実行しています。O/Rマッパー自体に、便利なメソッドが定義してあったり、演算子のオーバーライドといった手法を活用して、SQL文字列を組み立て、適切にデータベースと交信を行うような作りになっているわけです。

図01 O/Rマッパーの働き



10-02 シンプルなO/Rマッパーを作る

O/Rマッパーは一見とても高度なもののように見えますが、内部ではそれほど難しいことをしているわけではありません。簡単な機能を持ったO/Rマッパーなら、Pythonのクラスやオブジェクト指向について基本的な知識を使えば、簡単に作ることができます。ここでは、Pythonでシンプルな機能を持つO/Rマッパーを作りながら、仕組みやはたらきについて学んでみましょう。

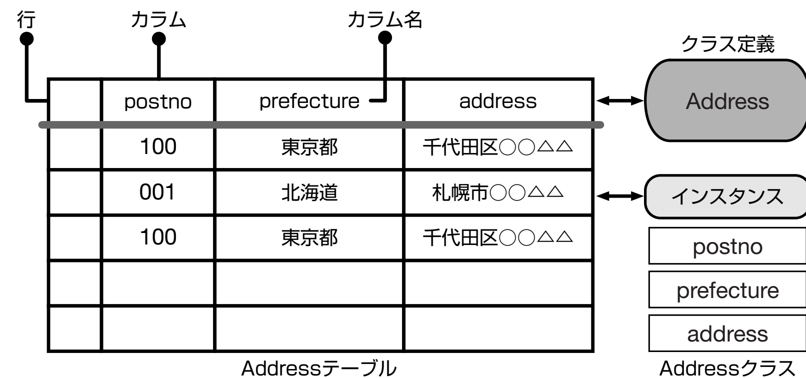
設計の指針

まずは簡単に、どのようなO/Rマッパーを作るかを決めることから始めましょう。

基本となる考え方は、**テーブルをクラスで表現し、データ(列)をインスタンスオブジェクトで表現する**ということです。

テーブルを表現するクラスには、テーブルのカラムの種類などを定義するようにします。テーブルとクラスが対応するので、1つのテーブルごとに1つのクラスができることになります。

図02 O/Rマッパーでは、テーブルがクラスに対応し、データがインスタンスに対応する



同じようなコードがいたるところに氾濫することを避けるため、テーブルを表現するクラスの親となるクラスを定義します。この親クラスの名前を**BaseMapper**としましょう。汎用的な処理を行うメソッドは、この親クラスに定義しておくことにします。この親クラスの機能は、必ず子供のクラスが継承した上で利用します。

このように、継承して利用することを前提に定義するクラスのことを**抽象クラス**と呼ぶことがあります。

データベース上のデータは、インスタンスオブジェクトのアトリビュートに保存することにします。データベース上で、データを保存しているテーブルを表現するクラスを元に、クラスインスタンスを作ることになります。

今回作るO/Rマッパーでは、以下の機能を実装することにします。

- クラスの定義を元にテーブルを作る機能
- テーブルにデータを追加(INSERT)する機能
- データを更新する機能(UPDATE)
- テーブルから条件に合うデータを取り出す機能(SELECT)

またBaseMapperクラスを定義するモジュール名は**simplemapper**とします。

テーブルを作成する

まずは、テーブルを作成するメソッドについて解説します。データベースでは、データを保存する前にテーブルを作っておく必要があります。O/Rマッパーを使うときには、このクラスを使って事前にテーブルを作っておくようにします。

テーブルを作るにはCREATE TABLE文を使用します。テーブルに含まれるカラムの名前とデータ型が分かっていればCREATE TABLE文に相当するSQL文字列を作ることができます。つまり、テーブルに含まれるカラムの情報から、テーブルを作るSQLを自動生成できる、ということです。

今回のO/Rマッパーでは、**rows**という決まった名前のクラスアトリビュートにテーブルに含まれるカラムの情報を保存しておくようにします。アトリビュートの内容はタプルのタプルです。カラム名とカラムのデータ型が並んだタプルを、カラムの数だけ並べるようにします。カラムの定義はテーブルごとに異なります。そのため、カラムの定義は抽象クラスを継承したクラス上で、以下のように行うことになります。

```
rows=(('foo', 'int'), ('bar', 'text'))
```

クラスに対応するテーブル名はクラス名からとることにします。また、テーブル上に登録された個々のデータを識別するため、**id**という名前のカラムを暗黙に追加するようにします。

以下が、抽象クラスに定義した、テーブルを作成するメソッドです。メソッド内で行っていることは、SQL文に相当する文字列の生成と、SQLを送信する処理となります。

List01 createtable()メソッドの実装(simplemapper.py)

```
    @classmethod                                              ❶
    def createtable(cls, ignore_error=False):
        """
        定義に基づいてテーブルを作る
        """
        sql="""CREATE TABLE %s (
                id INTEGER PRIMARY KEY, %s );"""
        columns=', '.join(["%s %s"%(k, v) for k, v in
        cls.rows])                                            ❷
        sql=sql%(cls.__name__, columns)
        cur=cls.getconnection().cursor()
        try:
            cur.execute(sql)
        except Exception, e:
            if not ignore_error:
                raise e
        cur.close()
        cls.getconnection().commit()
```

メソッドの定義をするdef文の上には、アットマークから始まる見慣れない構文があります(❶)。この構文は**デコレータ**と呼ばれています。Python 2.4から追加された比較的新しい構文です。デコレータは、関数やメソッドに特殊な働きを追加するためなどに利用されます。@classmethodという宣言は、メソッドを**クラスメソッド**として定義するために利用されます。

クラスメソッドはメソッドの一種ですが、通常のメソッドとは機能が異なっています。簡単に言うと、普通のメソッドはとクラスメソッドはメソッドの**持ち主**が異なります。普通のメソッドの持ち主はインスタンスですが、クラスメソッドの持ち主はクラス自体(クラスオブジェクト)となります。たいていのメソッドは持ち主を対象に処理を行いますので、持ち主が異なれば処理の対象も異なります。

Pythonのメソッドでは、第1引数にクラスインスタンスをとります。しかし、クラスメソッドでは第1引数にクラス自体(クラスオブジェクト)をとります。クラスメソッドは、「ins.clsmethod()」のようにインスタンスオブジェクトから呼び出すこともできますし、「Class.clsmethod()」のようにクラス自体から呼び出すこともできます。どちらの場合でも、第1引数に渡ってくるのはクラス自体のオブジェクトです。

なぜ、テーブルを作るメソッドをクラスメソッドとして定義する必要があるのでしょうか。このメソッドがどのような場面で利用されるか、このメソッドではどのような情報が必要かを考えると答えが見えてきます。

テーブルは、テーブルが存在しない状態で作ります。テーブルが存在しなければ、テーブル上にデータも存在しません。そのため、テーブルを作る前

にはO/Rマッパークラスのインスタンスは存在しないのです。この時点ではテーブル用のクラスしか存在しないため、クラスオブジェクトから呼び出せるメソッドを定義する必要があります。

また、テーブルを作成するSQL文字列を作るためには、クラスに定義された情報が必要です。そのため、メソッドでクラスオブジェクトを受け取った方が都合がよいのです。

メソッドの前半はSQL文字列を組み立てている部分です。このO/Rマッパーではクラス名をテーブル名とします。クラスオブジェクトの**__name__**というアトリビュートを使って、クラス名を取得しています。その後は、クラスオブジェクトに定義されたrowsというアトリビュートを使って、カラム名とデータ型を並べています(❷)。

SQL文字列が組み上がったら、カーソルオブジェクトを取得してデータベースにSQLを送信します。メソッドの引数にはフラグが渡ってきます。このフラグがTrueのとき、テーブル作成時に起こるエラー(例外)を無視します。テーブルがすでに存在したときに発生するエラー(例外)を無視し、**テーブルがあってもなくてもとにかく作ってみる**という処理を実現しています。

テーブルにデータを追加(INSERT)する

次に、テーブルにデータを追加するメソッドについて見てみましょう。テーブルにデータを追加するときには、INSERT文を使用します。INSERT文を作るときに必要な情報はテーブル名、カラム名、追加するデータ(カラムごとに必要)の3種類です。このうちテーブル名とカラム名はクラスオブジェクトから得ることができます。追加するデータは状況依存なので、引数としてメソッドで受け取ることにしましょう。

なお、このメソッドもデコレータを使ってクラスメソッドとして定義しています。データの登録はテーブルに対して行います。O/Rマッパーでは**テーブル=クラス**ですので、メソッドをクラスの持ち物として定義しています。

List02 insertメソッドの実装(simplemapper.py)

```
    @classmethod
    def insert(cls, **kws):
        """
        データを追加し,IDを返す
        """
        sql="""INSERT INTO %s(%s) VALUES(%s)"""
        rownames=', '.join([v[0] for v in cls.rows])
        holders=', '.join(['?' for v in cls.rows])
        sql=sql%(cls.__name__,rownames, holders)
        values=[kws[v[0]] for v in cls.rows]          ❶
        cur=cls.getconnection().cursor()
        cur.execute(sql, values)
        cur.execute("SELECT max(id) FROM %s"%cls.__name__)
        newid=cur.fetchone()[0]
        cls.getconnection().commit()
        cur.close()
        return newid
```

このメソッドでも、前半でSQL文字列の組み立てを行っています。テーブル名となるクラス名は__name__アトリビュートから、カラム名はrowsアトリビュートから取得できます。

登録を行うデータは、メソッドに引数として渡します。どのカラムにどのデータを登録したいかは、引数名で指定します。たとえば、次のようにメソッドを呼び出したとします。

```
ORClass.insert(foo=1, bar='test')
```

この場合は、fooというカラムに数値の1を、barというカラムに文字列の"test"を登録することになります。

メソッドの定義には****kws**というアスタリスクが2つ付いた引数が見えます。関数やメソッドにこのような引数が定義されていると、**任意のキーワード引数**を受け取れるようになります。受け取ったキーワード引数は辞書の形式で渡ってきます。つまり、次のようなメソッド呼び出しを行った場合は、

```
insert(foo=1, bar=2)
```

kwsという引数に「{'foo':1, 'bar':2}」という内容の辞書が代入されます。

メソッドの内部では、kwsという辞書とカラムの情報が入ったrowsアトリビュートを比較して、SQL文字列を組み立てています(❶)。SQL文字列の組

み立てが終わったら、カーソルオブジェクトを取得してデータベースにSQLを送信します。

インスタンスオブジェクトの初期化

以下のコードが、O/Rマッパーの抽象クラスの宣言と、データベースへのコネクションオブジェクトを管理するためのメソッドの定義です。テーブルごとにカラムを定義するためのアトリビュート（rows）があります。このアトリビュートには、カラム名とカラムの型をタプルにして並べます。基底クラスそのものが利用されることはないので、ここでは空のタプルが代入されています。基底クラスを継承したクラスで、同名のアトリビュートを定義する必要があります。

List03 BaseMapperクラスの宣言とコネクションメソッド（simplemapper.py）

```
class BaseMapper(object):
    """
    シンプルな機能を持つO/Rマッパーのベースクラス
    """
    rows=()

    connection=sqlite3.connect(':memory:')

    @classmethod
    def setconnection(cls, con):
        cls.connection=con

    @classmethod
    def getconnection(cls):
        return cls.connection
```

ところで、O/Rマッパーでは、クラスから作られるインスタンスオブジェクトはどのように生成されるでしょうか。O/Rマッパーの設計にもよりますが、インスタンスの生成のされ方はおおまかに2種類に分かれます。1つは、データベースに新しいデータを登録する意味でインスタンスを作る場合。インスタンス生成時には、引数として登録するデータを渡すようになるはずです。もう1つは、データベースにすでに登録されているデータを元に、インスタンスを生成する場合です。インスタンス生成時には、既存データ1つを特定するための情報（ID）を引数に渡します。

Pythonのクラスでは、__init__()という初期化メソッドでインスタンスの初期化を行います。O/Rマッパーの場合は、どのような目的でインスタンスを得たいのかによって、初期化の手法が分かれる、ということになります。メソッドには、引数として情報を渡すことができます。この引数を使って、新しいデータを登録するのか、既存データを参照するのかを切り分けることができるはずです。

既存データをデータベースから引き出してインスタンスオブジェクトを作る場合、どのデータを使うかを特定するための情報をメソッドに渡す必要があります。いま作っているO/Rマッパーでは、それぞれのデータを判別するためにidというカラムを追加しています。このidを引数に渡したときに、既存データからインスタンスを作る、という場合分けをすることにしましょう。id以外の引数が渡されていたら、新規登録を意味することになります。

以下のコードが、O/Rマッパーの初期化メソッドの定義です。

List04 __init__メソッドの実装（simplemapper.py）

```
    def __init__(self, **kws):
        """
        クラスを初期化する
        idを引数に渡された場合は,既存データをSELECTして返す
        その他のキーワード引数を渡された場合は,データをDBに
        Insertする
        """
        if 'id' in kws.keys():                                    ❶
            rownames=[v[0] for v in self.__class__.rows]
            rownamestr=', '.join(rownames)
            cn=self.__class__.__name__
            sql="""SELECT %s FROM %s WHERE id=?"""%(rownamestr,
            cn)
            cur=self.getconnection().cursor()
            cur.execute(sql, (kws['id'],))
            for rowname, v in zip(rownames, cur.fetchone()):     ❷
                setattr(self, rowname, v)
            self.id=kws['id']
            cur.close()
        elif kws:
            self.id=self.insert(**kws)                            ❸
            rownames=[v[0] for v in self.__class__.rows]
```

```
            for k in kws.keys():
                if k in rownames:
                    setattr(self, k, kws[k])
```

初期化メソッド（__init__()）の内容を簡単に見てみましょう。このクラスの初期化メソッドはいろいろな種類の引数を受け付ける必要があるため、アスタリスクを2つ先頭に持つキーワード引数を定義しています。

メソッドの中では、引数の種類を判断して、処理を振り分けています（❶）。

もし、引数名の中にidという名前が見つかったら、既存のデータからインスタンスオブジェクトを作ります。ここではSELECT文に相当するSQL文字列を組み立ててデータベースに送信し、データを得ています。

データベースから取得したデータは、カラム名に相当するインスタンスのアトリビュートにデータを代入します。カラム名自体は文字列として与えられます。そのため、ここではsetattr()という関数を使ってインスタンスオブジェクト（self）にアトリビュートを設定しています（❷）。

もし、idという名前を持つ引数が見付からなかった場合は、クラスメソッドのinsert()を使ってデータをデータベースに登録します（❸）。その後、メソッドに渡された引数を元にアトリビュートを設定します。

データを更新する機能（UPDATE）

次に、データを更新するメソッドについて解説しましょう。このO/Rマッパーの場合は、データベースから取得したデータをインスタンスオブジェクトとして表現します。O/Rマッパーを使うプログラム側では、インスタンスのアトリビュートを通じてデータにアクセスします。アトリビュートを参照したり、必要があればアトリビュートのデータを更新します。

データを更新した場合は、更新した内容をデータベースに反映する必要があります。そのときに呼ばれるのがupdate()メソッドです。プログラム側では、インスタンスを一通り操作した後、メソッドを呼び出して更新内容を保存することになります。

__setattr__()という特殊メソッドを定義すると、アトリビュートを更新したらすぐ、データベースに更新内容を反映するような仕組みを作ることもできます。このO/Rマッパーは機能を最小限度にとどめているため、そのような機能は実装していません。

List05 update()メソッドの実装（simplemapper.py）

```
    def update(self):
        """
        データを更新する
        """
        sql="""UPDATE %s SET %s WHERE id=?"""
        rownames=[v[0] for v in self.__class__.rows]
        holders=', '.join(['%s=?'%v for v in rownames])
        sql=sql%(self.__class__.__name__, holders)
        values=[getattr(self, n) for n in rownames]
        values.append(self.id)
        cur=self.getconnection().cursor()
        cur.execute(sql, values)
        self.getconnection().commit()
        cur.close()
```

このメソッドも、処理のほとんどはSQL文字列を組み立てる処理を実行しています。データベース上のデータを更新するためのUPDATE文を組み立てるために必要な情報はすべてクラスインスタンスが持っています。テーブル名はクラスオブジェクトから得ることができますし、カラムの名前はrowsというクラスのアトリビュートから得ることができます。更新に利用するデータはインスタンスオブジェクトのアトリビュートが持っています。このような文字列を元に、SQL文字列を組み立てて、データベースに送信してデータの更新を行っています。

テーブルから条件に合うデータを取り出す機能（SELECT）

データベース上のテーブルからデータを取り出すには、SELECT文というSQLを利用します。SELECT文には、テーブル名やデータを取り出すカラム名の他に、取り出すデータの条件を指定することができます。条件はWHERE句に記述します。**AND**や**OR**を使った複雑な条件を指定することも可能です。以下にWHERE句を使ったSQLの例を示します。

```
SELECT foo, bar FROM testtable WHERE foo > 2 AND foo < 100;
```

WHERE句に含まれる条件を考えると、テーブルからデータを取り出すために必要なSQL文字列にはとても多くのバリエーションがあることが分かります。この条件をうまく扱い、テーブルからデータを取り出すことができるメソッドがO/Rマッパーにあると、とても便利に利用できそうです。

メソッドに何らかの情報を渡したいときには**引数**を使います。この引数を使って、データを取り出すときの条件を指定する方法があれば、そのようなメソッドが作れそうです。

Pythonには**キーワード引数**という機能があります。この機能を使うと、メソッドや関数で任意の引数を受け取ることができます。この機能を使って、データベースからデータを選択する条件を指定できそうです。たとえば、「fooという名前のカラムが数値の1である」という条件は、次のように表現できます。

```
ins.select(foo=1)
```

ただし、このままでは「等しくない(!=)」や「より大きい(>)」というような条件が指定できません。そこで、引数の名前付けのルールを少々拡張することにしましょう。カラム名の後に**_ne**という文字列を追加すると「!=」という条件にし、また**_gt**という文字列を追加すると「>」という条件が指定されているものと解釈します。複数条件が指定されていたら、AND条件として扱うことにします。

TestTableというO/Rマッパーのクラスがあり、ここから**foo > 2 and foo <= 100**というWHERE句に相当するデータを選択するときには、以下のようなメソッド呼び出しを行うことになります。

```
TestTable.select(foo_gt=2, foo_lte100)
```

以下に、テーブルからデータを選択するselect()メソッドの定義を示します。

なお、テーブルからデータを選択する処理は、テーブルを対象に行います。そのため、データを選択するメソッドもクラスメソッドとして実装しています。

List06 select()メソッドの実装(simplemapper.py)

```
where_conditions={                                      ❶
    '_gt':'>', '_lt':'<',
    '_gte':'>=', '_lte':'<=',
    '_like':'LIKE' }

@classmethod
def select(cls, **kws):
    """
    テーブルからデータをSELECTする
    """
    order=''
    if "order_by" in kws.keys():
        order=" ORDER BY "+kws['order_by']
        del kws['order_by']
    where=[]
    values=[]
    for key in kws.keys():
        ct='='
        kwkeys=cls.where_conditions.keys()
        for ckey in kwkeys:
            if key.endswith(ckey):
                ct=cls.where_conditions[ckey]
                kws[key.replace(ckey, '')]=kws[key]
                del kws[key]
                key=key.replace(ckey, '')
                break
        where.append(' '.join((key, ct, '? ')))
        values.append(kws[key])
    wherestr="AND ".join(where)
    sql="SELECT id FROM "+cls.__name__
    if wherestr:
        sql+=" WHERE "+wherestr
    sql+=order
    cur=cls.getconnection().cursor()
    cur.execute(sql, values)
    for item in cur.fetchall():
        ins= cls(id=item[0])
        yield ins                                       ❷
    cur.close()
```

メソッドの大部分は、キーワード引数からWHERE句に相当するSQL文字列を作っている部分です。_gtや_nなど引数名の末尾に付ける文字列は辞書として定義してあります（❶）。対応する比較演算子を辞書の値としてあらかじめ定義しておくわけです。

その後、キーワード引数の名前を見ながら、WHERE句の条件に相当する文字列を組み立てていきます。_gtなど特別な働きを持つ文字列が末尾にあるときは、末尾の文字列を取り除いた上で比較用の文字列を作っていきます。

order_byという引数を追加すると、取り出すデータの並び順を指定できます。SQLには**ORDER BY句**があり、この句を利用します。

WHERE句に相当する文字列ができ上がったら、カーソルオブジェクトを使ってデータベースにSQLを送信、結果を得ます。得た結果から、O/Rマッパークラスのインスタンスオブジェクトを作って返します。

O/Rマッパークラスのインスタンスを返すときには、return文の代わりにyield文を使います（❷）。プログラムの書き方としては、見つかったインスタンスを1つずつ返しているように見えます。メソッドの呼び出し元では、**イテレータ**として処理します。このようにyield文を使って戻り値を返すメソッドや関数は**ジェネレータ**と呼ばれています。

そのほかの処理

O/Rマッパーの機能とは関係ありませんが、このBaseMapperクラスに、次の特殊メソッド__repr__()を定義しておきます。このメソッドは、インスタンスオブジェクトの概要を簡易に表示するためのもので、動作チェックで使用します。

List07 __repr__()メソッド（simplemapper.py）

```
    def __repr__(self):
        """
        オブジェクトの文字列表記を定義
        """
        rep=str(self.__class__.__name__)+':'
        rownames=[v[0] for v in self.__class__.rows]
        rep+=', '.join(["%s=%s"%(x, repr(
                getattr(self, x))) for x in rownames])
        return "<%s>"%rep
```


これでBaseMapperクラスは完成です。すべてのコードを1つにして「simplemapper.py」というファイル名で、Pythonのインストールディレクトリに保存してください。

10-03 O/Rマッパーの利用例

では、実査にO/Rマッパーを使ってみましょう。インタラクティブシェルを使って、O/Rマッパー用クラスを使ったコードを簡単に書いてみます。

まず、BaseMapperクラスをインポートし、抽象クラスを継承したクラスを定義する必要があります。テーブルに定義するカラムは、クラスアトリビュートにタプルとして定義します。数値とテキストだけを持つシンプルなテーブルです。その後、クラスメソッドを使ってテーブルを作ります。

```
>>> from simplemapper import BaseMapper
>>> class TestORClass(BaseMapper):
...     rows=(('num', 'int'), ('body', 'text'))
...
>>> TestORClass.createtable()
```

テーブルを作ったら、テーブルにデータを追加します。10回のループを組み、追加するデータを引数として与えます。range（10）というシーケンスを与えて0から9までの数値を繰り返し変数に代入しながらループを実行し、繰り返し変数に合わせて異なったデータをテーブルに追加しています。

```
>>> for i in range(10):
...     ins=TestORClass(num=i, body='body'+str(i))
...
```

データを追加したあと、今度は追加したデータをデータベースから取り出して画面に表示しています。取り出した結果はクラスのインスタンスオブジェクトとして返ってきますが、BaseMapperクラスに定義された__repr__()メソッドによりインスタンスオブジェクトの内容がわかりやすく表示されるはずです。

アトリビュートを書き換えて、インスタンスのupdate()メソッドを呼び出すことで、変更内容をデータベースに更新しています。

```
>>> ins=TestORClass(id=1)
>>> ins
<TestORClass:num=0, body=u'body0'>
>>> ins.num=100
>>> ins.body=u'body100'
>>> ins.update()
```

最後に、追加したデータ、更新したデータを確認する意味で、データベースからデータを取り出してみましょう。**numが5以上**という条件を与えます。引数の最後に**_gt**という文字列を追加します。結果を表示して、正しいデータが取り出されているかどうか確認してみましょう。先ほどデータを更新しています。更新したデータも表示されるはずです。

```
>>> for ins in TestORClass.select(num_gt=5):
...     print ins
...
<TestORClass:num=100, body=u'body100'>
<TestORClass:num=6, body=u'body6'>
<TestORClass:num=7, body=u'body7'>
<TestORClass:num=8, body=u'body8'>
<TestORClass:num=9, body=u'body9'>
```

今回作ったO/Rマッパーはたかだか140行ほどの短いプログラムです。これだけの短いコードでも、データベースの処理をPython的な方法で行い、プログラムからSQL文字列を追い出すことができるわけです。

O/Rマッパーを活用すると、よりシンプルで見通しがよいプログラムを作れます。そのぶん、より複雑なプログラムを、より手軽に作れるようになるわけです。

CHAPTER 11

RSSリーダーを作る その2

テンプレートエンジンとO/Rマッパーという強力な2つの武器を手に入れました。この2つの仕組みを使えば、Webアプリケーションの開発がずっと効率的になります。より少ない手数で、より高機能なアプリケーションを開発できるようになるわけです。

ここでは、2つの仕組みを活用して、P.86で作ったRSSリーダーの機能強化をしてみましょう。

P.86で作ったRSSリーダーは、RSSのURLを直接打ち込み、RSSの一覧を表示する機能しか持っていませんでした。今回は、RSSを登録する機能を追加します。RSSが複数登録されていたら、複数のRSSを巡回してRSSの内容を表示するようにします。

RSSの登録や修正をするためにはユーザインターフェースが必要です。フォームを使って、RSSを新規登録したり、既存のRSSを修正するような仕組みを作るわけです。このような機能は、Webアプリケーションではとても一般的な機能です。今回は、RSSリーダーの拡張に合わせて、フォームを使って項目を登録、編集する仕組みをどのように作るかについても解説します。

11-01 RSSリーダーの機能追加

まず最初に、どのような機能追加を行うかについて決めることから始めましょう。今回目指すのは、**複数RSSへの対応**です。巡回するRSSを登録できるようにし、複数RSSがある場合は複数のRSSを巡回して、結果を表示します。

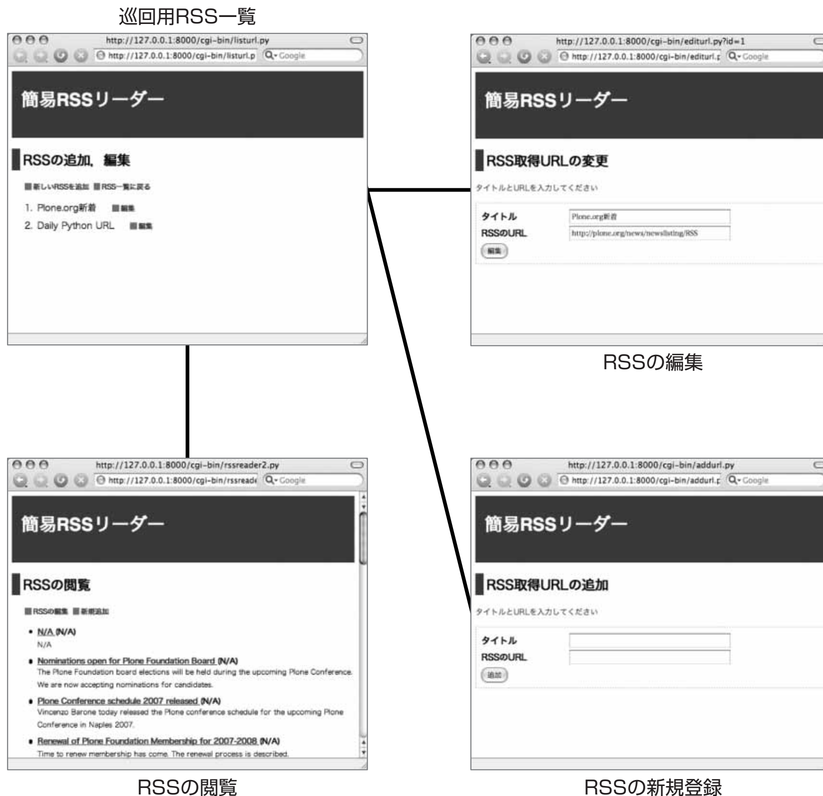
登録するRSSのURLなどは、データベース（SQLite3）に保存します。O/Rマッパーの抽象クラスを継承したクラスを新たに作り、データベースに関わ

る処理に利用します。

RSSを登録できるようにするためには、RSSの内容を入力し、Webアプリケーションに届けるためのUIが必要です。RSS登録用のUIには2種類あります。1つはRSSを新規に登録するためのUIです。もう1つは、登録済みのRSSを変更するためのUIです。フォームを表示するHTMLを、テンプレートエンジンを活用して作成します。Webアプリケーション側でフォームからの入力を受け取り、必要に応じて処理をします。Webアプリケーションの処理は、**RSSの新規登録**と**既存RSSの変更**という2種類が必要になります。

また、RSSの追加、変更をするとき、不正なデータが入力されたらエラーを表示して追加、登録を受け付けないようにする仕組みを作ります。このような仕組みをバリデーションチェックと呼ぶことはすでに解説しました。

図01 今回のRSSリーダーは、いくつかの画面で構成される



前回作ったRSSリーダーでは、RSSのURL入力フォームと、巡回結果を表示するプログラムが1つでした。つまり、1枚の画面ですべての処理が実行できたのです。今回の拡張によって、RSSリーダー全体で複数の画面を持つことになります。図のような画面構成になるはずです。1つの画面ごとに、1つのプログラムが必要になります。プログラムだけでなく、表示用のテンプレートファイルも必要です。今回作るWebアプリケーションは、複数のファイルで構成されることになります。

O/Rマッパーのクラスを作る

巡回用のRSSを登録するために、Pythonのクラスを作りましょう。O/Rマッパーの抽象クラスを継承して、巡回用RSSのデータを保存するためのクラスを定義します。O/Rマッパーの抽象クラスを継承することによって、クラスを利用するだけでデータベースに対するアクセスができるようになります。クラスには、テーブルに定義するカラムを定義しておきます。

以下がクラスを定義するためのスクリプトファイルです。cgi-binフォルダに設置します。驚くほど短いスクリプトファイルですが、これでもちゃんと機能します。

List01 rssurl.py

```
#!/usr/bin/env python
# coding: utf-8

import sqlite3
from os import path
from simplemapper import BaseMapper

class Rssurl(BaseMapper):                                  ❶
    rows=(('title', 'text'), ('url', 'text'))

p=path.join(path.dirname(__file__), 'urls.dat')
con=sqlite3.connect(p)
BaseMapper.setconnection(con)

Rssurl.createtable(ignore_error=True)
```

前半はモジュールをインポートするなどの準備を行っている部分です。クラス定義は中ほどにある2行だけで、巡回用RSSのデータを保存するテーブルのカラムを定義しています（❶）。その後では、SQLite3へのコネクションオブジェクトを設定し、テーブルが存在しなかったら作成しています。SQLiteのデータベースファイルは、このスクリプトと同じフォルダの、「urls.dat」という名前で保存されることになります。

スクリプトが完成したら、このファイルをcgi-binフォルダに保存しておきます。

動作チェック

スクリプトを設置したら、このクラスを使って簡単なテストをしてみましょう。シェルでcgi-binフォルダに移動し、Pythonのインタラクティブシェルを起動します。以下のコマンドを入力してみてください。

●動作チェック

```
>>> from rssurl import Rssurl
>>> ins=Rssurl(title='Daily Python URL',
...     url='http://www.pythonware.com/daily/rss.xml')
>>> for rss in Rssurl.select():
...     print rss
...
<Rssurl:title=u'Daily Python URL', url=u'http://www.pythonware.
com/daily/rss.xml'>
```

この例では、RssurlクラスというO/Rマッパークラスを使ってデータベースにテスト用のデータを登録しています。登録しているのは、Pythonの新着ニュース（英語）を配信している「Daily Python URL」というサイトのRSSです。Rssurlクラスのインスタンスを生成してデータを登録したあと、select()メソッドを呼んで登録した内容を確認しています。

動作チェックが終了したら、rssurl.pyとurls.dat、およびP.172で作成したsimplemapper.pyをcgi-binフォルダの中にコピーしておいてください。

巡回用RSSの一覧ページを作る

次に、登録済みの巡回用RSSをリスト表示するプログラムを作ります。

まずは、動的な出力を行うPythonのプログラムを作ります。「listurl.py」という名前で、cgi-binフォルダに保存します。

先ほど定義したRssurlという名のO/Rマッパークラスを使って、データベースから登録済みの巡回用RSSを取り出し、テンプレートエンジンに渡す、というのが主な処理です。

List02 listurl.py

```
#!/usr/bin/env python
# coding: utf-8

from simpletemplate import SimpleTemplate
from rssurl import Rssurl
from os import path
from httphandler import Request, Response
import cgitb; cgitb.enable()

value_dic={'rsslist':[x for x in Rssurl.select(order_by='id')]} ❶

res=Response()
p=path.join(path.dirname(__file__), 'urllist.html')
t=SimpleTemplate(file_path=p)
body=t.render(value_dic)
res.set_body(body)
print res
```

テンプレートエンジンで登録済みのRSSを表示するため、**rsslist**というキーにリストを渡しています（❶）。キーに対応する値には、O/Rマッパークラスのインスタンスオブジェクトをリストにして登録します。クラスのselect()メソッドでデータベースからすべてのデータを取り出し、結果をリスト内包表記でリストに変換しています。たった1行のコードでこれだけの処理を実現しています。

テンプレートエンジンに渡す辞書を作ったら、次はResponseオブジェクトを作ります。テンプレートのパスを指定してテンプレートエンジンのインスタンスを作り、文字列に変換して結果として出力しています。

テンプレートファイルの作成

次に、リストを表示するためのテンプレートファイルを作ります。UTF-8

のエンコーディングで、cgi-binフォルダに設置します。

テンプレートエンジンが使えるようになり、Webアプリケーションの出力として使うHTMLの自由度が上がったので、デザインにも凝ってみることにします。CSSを使って、見栄えのする画面を作ってみることにしましょう。

List03 urllist.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-
8" />
<link rel="stylesheet" href="/style.css" type="text/css"/>
</head>
<body>
<h1 class="header">簡易RSSリーダー</h1>
<h2 class="title">RSSの追加、編集</h2>
<div class="control">
    <a href="./addurl.py">新しいRSSを追加</a>
    <a href="./rssreader2.py">RSS一覧に戻る</a>
</div>
<ol>
$for item in rsslist:                                          ❶
    <li>
        ${item.title}
        <span class="control">
        <a href="./editurl.py?id=${item.id}">編集</a>
        </span>
    </li>
$endfor
</ol>
</body>
</html>
```

このテンプレートでは、テンプレートエンジンを利用しているPythonのプログラムからrsslistというシーケンス（リスト）を受け取っています。シーケンスには、データベースからO/Rマッパーを使って得た巡回用RSSのオブジェクトがすべて入っています。受け取ったシーケンスを使い、テンプレートのループ構文$for～：を使って繰り返し処理を行っています（❶）。

ループの内部では、<a>タグを使って編集用のURLへのリンクを埋め込んでいます。編集用のプログラムは「editurl.py」という名前にする予定です。リンクには、テンプレートエンジンの埋め込み記法を使い、巡回用RSSを特定するための情報（id）を埋め込んでいます。GETメソッドを使って、編集する巡回用RSSを特定するための情報を渡しているわけです。

またこのテンプレートでは見栄えをよくするためにCSSを適用していますが、ここでは内容を解説しません。本書のサポートサイトでダウンロード可能なサンプルファイルを参照してください。

先ほど、インタラクティブシェルを使ってテスト用のデータを1件登録してあります。P.44で作成したcgiserver.pyを実行してPython製のWebサーバを走らせ、Webブラウザでこのプログラム（cgi-bin/listurl.py）にアクセスをすると、テストとして登録したデータが表示されます。

図02 登録済みの巡回用RSSをリスト表示する画面

編集用フォームを作る

次に、巡回用RSS編集用のフォーム、およびデータを更新するプログラムを作りましょう。このフォームは、先ほど作ったリストからリンクされています。編集したい項目のリンクをクリックして、編集用のフォームを表示し、内容を編集する、という流れになります。

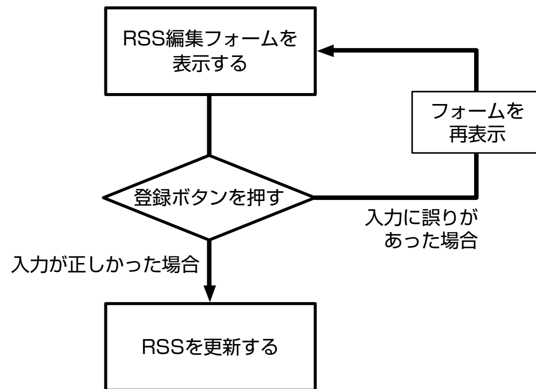
リスト上のリンクには、編集するデータを特定するための情報が埋め込まれています。GETメソッドを使って、URLにデータベース上のデータを特定するためのidを埋め込みます。プログラム側では、このidを読み取って編集

対象となるデータを特定します。

フォームには、不正なデータが入力されることがあるかもしれません。たとえば、正しいURLが入力されていない場合、そのままデータを登録するとRSSが正しく読み込めません。そのようなことを避けるため、不正なデータが入力されたときにはフォームを再表示し、正しいデータを入力するように求めるような動作にします。不正な値が入力された場合には、フォームの横に警告の表示をするようにします。

データを変更するときの流れは次の図のようになります。このように入力や出力の流れを表した図を**遷移図**と呼ぶことがあります。

図03 編集フォームの遷移図



以下が、巡回用URLを編集するためのプログラム部分となります。フォームの表示とデータの更新を1つのプログラムで行っています。

List04 editurl.py

```
#!/usr/bin/env python
# coding: utf-8

from simpletemplate import SimpleTemplate
from rssurl import Rssurl
from os import path
from httphandler import Request, Response

import cgitb; cgitb.enable()

errors={}
value_dic={'errors':errors, 'title':'', 'url':'', 'item_id':'' }

req=Request()
f=req.form

p=path.join(path.dirname(__file__), 'editform.html')

if not f.getvalue('posted'):
    id=f.getvalue('id')
    rss=Rssurl(id=int(id))
    value_dic.update({'title':rss.title, 'url':rss.url,
                      'item_id':id})
else:
    id=f.getvalue('id')
    title=unicode(f.getvalue('title', ''), 'utf-8', 'ignore')
    url=unicode(f.getvalue('url', ''), 'utf-8', 'ignore')
    value_dic.update({'title':title, 'url':url, 'item_id':id})
    if not title:                                                    ❶
        errors['title']=u'タイトルを入力してください'
    if not url.startswith('http://'):
        errors['url']=u'正しいURLを入力してください'
    if not errors:
        rss=Rssurl(id=int(f.getvalue('id')))
        rss.title=f.getvalue('title')
        rss.url=f.getvalue('url')
        rss.update()
        p=path.join(path.dirname(__file__), 'posted.html')
        value_dic['message']=u'RSS取得URLを編集しました'

t=SimpleTemplate(file_path=p)
res=Response()
body=t.render(value_dic)
res.set_body(body)
print res
```

フォームを表示する場合には、編集したいデータの既存の値を埋め込んで表示します。既存の値は、**value_dic**という辞書にしてテンプレートに渡します。入力にエラーがあった場合には、辞書の**errors**というキーに辞書を登

録して渡します。テンプレート側では、辞書の内容を見て必要に応じて情報を埋め込んでいます。

プログラムの後半以降、トップレベルにあるelseブロックでは、POSTされたデータの処理をしています。もし不正な値があった場合にはエラーメッセージを辞書に登録します。もし正しいデータがPOSTされた場合には、O/Rマッパーのクラスを使ってデータを更新しています。

次に、データ更新用のフォームを表示するためのテンプレートを作ります。プログラムから受け取った辞書の内容を使って埋め込みを行う、というのが基本的な動作です。条件分岐（$if）を使って、エラーがある場合にのみエラーの表示をしています（❶）。

List05 editform.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
    <link rel="stylesheet" href="/style.css" type="text/css"/>
</head>
<body>
<h1 class="header">簡易RSSリーダー</h1>
<h2 class="title">RSS取得URLの変更</h2>
<p class="description">タイトルとURLを入力してください</p>
<form method="post" action="editurl.py">
    <label for="title">タイトル</label>
    <input type="text" name="title" size="40"
                     value="${title}" />
    $if errors.has_key('title'):
        <span class="error">${errors.get('title')}</span>
    $endif
    <br clear="all"/>
    <label for="url">RSSのURL</label>
    <input type="text" name="url" size="40" value="${url}" />
    $if errors.has_key('url'):
        <span class="error">${errors.get('url')}</span>
    $endif
    <br clear="all"/>
    <input type="hidden" name="posted" value="1" />
    <input type="hidden" name="id" value="${item_id}" />
    <input type="submit" value="編集" />
```

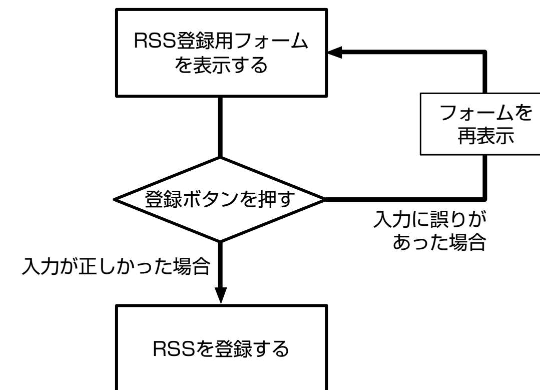
```
</form>
</body>
</html>
```

新規登録用フォームを作る

更新用フォームの次は、巡回用RSSの新規登録を行うフォームを作りましょう。入力値のチェックが必要であることなど、新規登録用フォームの機能や働きは更新用フォームにとてもよく似ています。たいていのWebアプリケーションでは、データの追加と更新を行う部分はとてもよく似通った処理になります。

ただし、データを新しく登録するわけですから、初期状態のフォームにはなにも表示しません。また、同じURLを持つURLを複数登録しても無意味です。入力値のチェックとして、登録済みのURLと同じであればエラーとみなす、というチェックを加えましょう。

図04 新規登録フォームの遷移図



以下が、巡回用URLを新規登録するためのプログラム部分となります。更新フォームと同じく、フォームの表示とデータの追加を1つのプログラムで行っています。

≡**List06** addurl.py

```
#!/usr/bin/env python
# coding: utf-8

from simpletemplate import SimpleTemplate
from rssurl import Rssurl
from os import path
from httphandler import Request, Response
import cgitb; cgitb.enable()

errors={}
value_dic={'errors':errors, 'title':'', 'url':'', 'item_id':''}

req=Request()
f=req.form

if f.getvalue('posted'):
    title=unicode(f.getvalue('title', ''), 'utf-8', 'ignore')
    url=unicode(f.getvalue('url', ''), 'utf-8', 'ignore')
    value_dic.update({'title':title, 'url':url})
    if not title:
        errors['title']=u'タイトルを入力してください'
    if not url.startswith('http://'):
        errors['url']=u'正しいURLを入力してください'
    if [x for x in Rssurl.select(url=url)]:                    ❶
        errors['url']=u'このURLは登録済みです'
    if not errors:
        Rssurl(title=title, url=url)
        p=path.join(path.dirname(__file__), 'posted.html')
        value_dic['message']=u'RSS取得URLを追加しました'

res=Response()
p=path.join(path.dirname(__file__), 'addform.html')
t=SimpleTemplate(file_path=p)
body=t.render(value_dic)
res.set_body(body)
print res
```

トップレベルにあるifブロックの内部で、データのチェックと登録を行っています。入れ子になったifブロックの3番目では、すでに登録されているURLとの重複チェックを行っています（❶）。リスト内包表記を使ったたった1行の処理で重複チェックが行えているわけです。O/Rマッパーを使うと、Python的な簡潔な記法でデータベースの処理ができ、直感的なプログラムが書けるようになります。

更新用のプログラムと同じように、辞書の形式でテンプレートに値を渡します。POSTされた値にエラーがあった場合には、**errors**というキーに辞書を渡し、エラーの内容を表示しています。

では次に、追加用フォームを表示するためのテンプレートを作りましょう。このテンプレートは更新用のフォーム（editform.html）とほぼ同じです。フォームのactionアトリビュートとボタンの名前が異なるくらいです。

≡**List07** addform.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-
8" />
<link rel="stylesheet" href="/style.css" type="text/css"/>
</head>
<body>
<h1 class="header">簡易RSSリーダー</h1>
<h2 class="title">RSS取得URLの追加</h2>
<p class="description">タイトルとURLを入力してください</p>
<form method="post" action="./addurl.py">
    <label for="title">タイトル</label>
    <input type="text" name="title" size="40" value="${title}" />
    $if errors.has_key('title'):
        <span class="error">${errors.get('title')}</span>
    $endif
    <br clear="all"/>
    <label for="url">RSSのURL</label>
    <input type="text" name="url" size="40" value="${url}" />
    $if errors.has_key('url'):
        <span class="error">${errors.get('url')}</span>
    $endif
    <br clear="all"/>
    <input type="hidden" name="posted" value="1" />
    <input type="hidden" name="id" value="${item_id}" />
    <input type="submit" value="追加" />
</form>
</body>
</html>
```

RSSの一覧ページを作る

登録、更新系の遷移が一通りできました。最後に、登録されているRSSを巡回し、一覧表示するプログラムを作ります。

RSSを取得し、Pythonのデータ型に変換する必要があります。これにはP.94で作った「rssparser.py」というモジュールが利用できます。RSSの取得に使うURLはデータベースに入っています。O/Rマッパーのクラスを使いデータを取り出し、RSSを順番に読み込んでいきます。読み込んだデータはシーケンスにして、テンプレートに渡します。テンプレートではループを使ってデータを表示します。

以下がRSSを巡回するプログラムです。前半ではモジュールのインポートを行っています。後半はテンプレートエンジンを使った表示を行っています。処理の本体は中ほどの2行のみです（❶）。O/Rマッパーを使ってデータベースからデータを取り出し、モジュールの関数を呼び出してRSSのデータを辞書のリストに変換しています。

なお、このプログラムでは、Webブラウザでアクセスしたときに毎回RSSを読み込んでいます。

List08 rssreader2.py

```
#!/usr/bin/env python
# coding: utf-8

from simpletemplate import SimpleTemplate
from rssurl import Rssurl
from os import path
from httphandler import Request, Response
from rssparser import parse_rss
import cgitb; cgitb.enable()

rsslist=[]
try:
    for rss in Rssurl.select(order_by='id'):                    ❶
        rsslist.extend(parse_rss(rss.url))
except:
    pass

res=Response()
p=path.join(path.dirname(__file__), 'rsslist.html')
t=SimpleTemplate(file_path=p)
body=t.render({'rsslist':rsslist[:20]})
res.set_body(body)
print res
```

以下がRSSの一覧を表示するためのテンプレートです。プログラムから受け取ったシーケンスを使ってループを実行し、必要な情報を表示しています。

List09 rsslist.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-
8" />
<link rel="stylesheet" href="/style.css" type="text/css"/>
</head>
<body>
<h1 class="header">簡易RSSリーダー</h1>
<h2 class="title">RSSの閲覧</h2>
<div class="control">
    <a href="./listurl.py">RSSの編集</a>
    <a href="./addurl.py">新規追加</a>
</div>
<ul>
    $for item in rsslist:
    <li>
        <dt>
            <a href="${item.get('link', '')}">
                ${item.get('title', '')}
            </a>
            (${item.get('pubDate', '')})
        </dt>
        <dd>
            ${item.get('description', '')}
        </dd>
    <li>
    $endfor
</ul>
</body>
</html>
```

「簡易RSSリーダー」に、任意のRSSを登録する機能を追加してみました。今回実装した、データの一覧や更新、追加という機能は、多くのWebアプリケーションが持つ機能です。フォームから登録されたデータが正しいかどうかを調べる処理を**バリデーション（妥当性）チェック**と呼びます。この機能も、今日的なWebアプリケーションによく搭載されています。使いやすいWebアプリケーションを作る上で、必須の機能です。

リクエストやレスポンスをクラスとして抽象化し、テンプレートエンジンやO/Rマッパーというような仕組みを使うことによって、Webアプリケーションの開発は効率的になってきました。しかし、実際により高度で気の利いた機能を持つWebアプリケーションを作ってみると、また冗長な部分、重複している部分が見えてきます。この後の章では、高度な機能の実装を助けるための仕組みについて考えてみたいと思います。

CHAPTER 12

バリデータとウィジェット―フォーム処理の抽象化

テンプレートエンジンやO/Rマッパーを作ることで、Webアプリケーションのプログラムがシンプルになりました。ここでは、高度で使いやすいWebアプリケーションに欠かせないフォームの処理をより手軽に作る方法について考えてみたいと思います。

12-01 バリデータの利用

データの入力を行うWebアプリケーションでは、入力値のチェック処理が欠かせません。フォームのようなUIに入力された値をプログラムの側でチェックし、期待される値が入力されているかどうかを確かめるわけです。このような処理は**バリデーション（妥当性）チェック**と呼ばれています。

たとえば、Webアプリケーションを使ってアンケートを集めるとします。アンケート項目と一緒にメールアドレスを入力してもらい、アンケート入力後に確認のメールを送るようにしたいとします。もちろん、入力された項目はデータベースに保存します。

WebアプリケーションのUIとなるフォームにはどのような文字列も入力できます。間違えて、メールアドレスとしてふさわしくない文字列が入力されるかもしれません。そのような値が登録されてしまうと、メールが送れなくなってしまいます。そうならないためにも、メールアドレスが文字列として正しいかを事前にチェックする必要があります。

また、Webのフォームは未入力の項目があっても送信できてしまいます。名前や住所のように、必ず入力してもらいたい項目があっても、空のままリクエストをPOSTできます。フォームに必須項目がある場合は、値が入力さ

れているかどうかを事前にチェックする必要があります。空項目のチェックも重要なバリデーションチェックの1つです。

メールアドレスのバリデーションチェック

バリデーションチェックという長く難しそうな名前が付いていますが、実際に行うことは文字列の検査にすぎません。リクエストのクエリにある文字列を調べて、正しい文字列で構成されているかを調べるわけです。

たとえば、数値だけ入力できる項目であれば**文字列が数字だけで構成されているかどうか**を調べます。文字列メソッドのisdigit()を使うことでそのような条件を検査できます。URLとして正しい文字列であることを簡易確認するためには、クエリ文字列が**「http://」で始まっているかどうか**を調べればよいでしょう。文字列の最初を調べるには、startswith()というメソッドを利用します。

メールアドレスのように、一定のパターンや条件を持った文字列を調べるには**正規表現**を使います。メールアドレスの形式や、どのような文字列が使えるのかについては「RFC 2822」という文書に細かく定義されています。厳密にメールアドレスを判別するのは実は難しいのですが、簡易に判別するのであれば正規表現を使うとよいでしょう。

インタラクティブシェルで試してみましょう。メールアドレスとして妥当な文字列をsearch()メソッドに渡すと、**Matchオブジェクト**が返ってきます。形式が整っていない、不正な文字列が混入しているなど、メールアドレスとして正しくない文字列の場合はNoneが返ってくるため、インタラクティブシェルにはなにも表示されません。

```
>>> import re
>>> pat=re.compile('[0-9a-z_&.+-]+@([0-9a-z-]+¥.)+[0-9a-z-]+$')
>>> pat.search('guidop@python.org')
<_sre.SRE_Match object at 0x1268e20>
>>> pat.search('foobarbaz')

>>> pat.search('foo@bar')

>>> pat.search('fooああ@bar.com')

>>>
```


メールアドレスだけでなく、URL相当の文字列を厳密に検査したい場合にも正規表現が利用できます。また、文字列がユニコードのカタカナだけで構成されているかどうか、といった条件も正規表現を利用して確認できます。

バリデータとは

文字列オブジェクトのメソッドや正規表現を利用すれば、入力値のバリデーションチェックは可能です。これまでのサンプルプログラムでは、そのような方法を使ってバリデーションチェックを行っていました。

しかし、このような方法でバリデーションチェックを行う場合、似たような処理が複数の部分に出てきて、コードが冗長になってしまう場合があります。バリデーションチェックにはいくつかの典型的なパターンがあります。空項目のチェック、数値に変換できる文字列かどうか、メールアドレスやURLとして正しい文字列かどうか、などがよく使われるパターンの例です。バリデーションチェックにはパターンがあるわけですから、クラスやモジュールとして実装しておけば、繰り返し利用できるようになります。コードの重複も防げるようになります。

バリデーションチェックの処理を抽象化し、典型的な処理を手軽に実行する目的で利用するのが**バリデータ**です。バリデータは、チェックを行う文字列などのオブジェクトを受け取り、期待通りの値であるかどうかを確かめて結果を返します。

12-02 バリデータを作る

バリデーションチェックの処理の実体は単純な文字列の検査です。バリデーションチェックの処理の内容はそれほど難しくなく、バリデータも比較的簡単に作れます。ここでは、簡単なバリデータを作ってみましょう。

まず、バリデータの設計について考えてみます。チェックすべき値を受け取り、正しい値かどうかを結果として返すのがバリデータに求められる機能です。このような単純な機能であれば関数として実装してもよさそうですが、のちのちのことを考えてクラスとして実装することにしましょう。バリデーションチェックを実行するためには、まずバリデータのクラスインスタンスを

作り、メソッドにチェックしたい値を渡して結果を受け取る、というような使い方をします。なお、メソッドには文字列のみを渡すことを想定します。

バリデータの戻り値と例外

Webアプリケーションでは、入力として文字列がプログラムに渡されます。プログラム内部では、文字列の入力を必要に応じて数値型などに変換する必要があります。どのデータをどのように変換するかは状況によってさまざまですが、バリデーションチェックに変換方法に関するヒントが隠されていることが多いのです。

たとえば、ある値について**文字列が数値だけで構成されているかどうか**をチェックしようとする場合を考えましょう。チェックした値は数値型として利用したい場合が多いはずです。このように、クエリ上のデータをどのように変換すべきかということをバリデータが知っている場合が多いのです。以上のようなことを踏まえて、バリデーションチェックに成功し、正しい値であることが分かった場合には、返り値として適切に変換したデータを返すようにしましょう。

最後に、バリデーションチェックに失敗したときにどうするかを考えましょう。変換に失敗したことを示すため、戻り値として**None**を返す、という方法も考えられます。ただしこの場合、バリデータを使う開発者が「Noneに特別な意味があるというルール」を覚える必要があります。戻り値のNoneに特別な意味がある、というのはPythonの世界で一般的なルールではありません。余計な知識を要求する仕様はよいとは言えませんので、この方法は見送ることにします。

要は**バリデーションチェックの最中に状態が変化した（値が不正だった）ことを関数やメソッドの外部に伝える**方法があればよいわけです。今回は戻り値を使わず、例外を使ってバリデーションチェックの失敗を外部に伝えるようにしましょう。バリデータを使う側は、必要に応じて例外を捕まえ、処理を続ければよいわけです。

バリデーションに失敗したときに利用する例外は、専用の例外を定義します。Pythonの例外の実体はクラスインスタンスです。Exceptionという、すべての例外の親となるクラスを継承して専用の例外を定義します。このように、専用の例外を定義することで、バリデーションの失敗だけを受け取って、特別な処理を実行することができるようになります。

List01 ValidationErrorクラス（validators.py）

```
#!/usr/bin/env python
# coding: utf-8

import re

class ValidationError(Exception):
    """
    バリデーションエラー用の例外クラス
    """
    def __init__(self, msg):
        Exception.__init__(self, msg)
        self.msg=msg

    def get_message(self):
        return self.msg
                    ⋮
```

バリデータの抽象クラスと初期化が不要なバリデータ

次にバリデータ用クラスの親となる抽象クラスを定義します。validate()というメソッドはバリデーションを行うために利用するメソッドです。Pythonには**抽象メソッド**のような機能がありません。そのため、継承した子クラスで必ず定義する必要があるメソッドを定義できません。しかし、メソッド名と引数の種類を明記しておくことで、継承したクラスがどのようなメソッドを実装すべきかが分かりやすくなるというメリットはあるはずです。

初期化メソッド__init__()がないのは、抽象的なバリデータにはインスタンスに保存しておくべきデータがないためです。

List02 BaseValidatorクラス（validators.py）

```
                    ⋮
class BaseValidator(object):
    """
    バリデータ用のベースクラス
    """
    def validate(self, value):
        return value
                    ⋮
```

抽象クラスを継承して、比較的処理が簡単な2つのバリデータクラスを定義してみましょう。1つは入力項目が空でないかどうかを調べるバリデータです。もう1つは、入力項目が整数値に変換できる文字列だけで構成されているかどうかを調べるバリデータです。どちらのバリデータも、バリデータの条件に合わない場合は例外をraiseする、という処理になっています。例外をraiseする際には、エラーの内容を文字列として渡しています。

どちらのバリデータも、インスタンスを作るときに前処理が必要ありません。そのため、初期化メソッドが定義されていません。

List03 バリデータクラス①(validators.py)

```
                    ⋮
class NotEmpty(BaseValidator):
    """
    項目が空でないことを調べるバリデータ
    """
    errors=(u'この項目は必須です。',)

    def validate(self, value):
        if not value:
            raise ValidationError(self.errors[0])
        return value


class IntValidator(BaseValidator):
    """
    項目が整数の数値であることを調べるバリデータ
    """
    errors=(u'この項目には数値を入力してください。',)

    def validate(self, value):
        try:
            value=int(value)
        except ValueError:
            raise ValidationError(self.errors[0])
        if int(abs(value))!=abs(value):
            raise ValidationError(self.errors[0])
        return value
                    ⋮
```


入力値が整数で、指定された範囲にあることを調べるバリデータ

日時や西暦のような数値を受け取るとき、入力値が一定の範囲内にあるかどうかを調べたいことがあります。そのような場合に利用できる汎用のバリデータを定義してみましょう。

バリデータが**正しい**と判断する数値の範囲は、状況によって異なります。最小値と最大値を、クラスのインスタンスオブジェクトを初期化するときに渡すようにすれば、インスタンスごとに異なる最小値と最大値を設定できます。

このようなバリデータでは初期化メソッドが必要になります。初期化メソッドでは、引数として最大値と最小値を受け取り、インスタンスオブジェクト(self)のアトリビュートに保存しています。インスタンスオブジェクトを作るときには**IntRangeValidator(1900, 2007)**のように引数に範囲を指定します。

List04 バリデータクラス②(validators.py)

```
                    ⋮
class IntRangeValidator(BaseValidator):
    """
    値が一定の範囲にあることを調べるバリデータ
    """
    errors=(u'入力された数値が設定された範囲を超えています。',)

    def __init__(self, min_val, max_val):
        self.min=min_val
        self.max=max_val

    def validate(self, value):
        value=IntValidator().validate(value)
        if value>self.max or self.min>value:
            raise ValidationError(self.errors[0])
        return value
                    ⋮
```

正規表現を使ったバリデータ：メールアドレス・URLのチェック

正規表現はバリデーションチェックのいろいろな場面で活用できます。イ

ンスタンスの初期化時に任意の正規表現パターンを受け取るバリデータがあれば、いろいろな場面で応用できるはずです。ここでは、正規表現を指定できるバリデータを定義してみましょう。

このバリデータでも初期化メソッドを定義します。インスタンスオブジェクトを作るときに、正規表現のパターン文字列を引数として受け取るようにします。初期化メソッドの内部では、受け取った正規表現文字列を元に正規表現オブジェクトを作り、インスタンスに保存します。

バリデーションチェックを行うvalidate()メソッドの内部では、初期化メソッドで作った正規表現パターンを使って入力値のチェックを行っています。

List05 正規表現を使ったベースクラス（validators.py）

```
                    ⋮
class RegexValidator(BaseValidator):
    """
    入力値が正規表現にマッチするかどうか調べるバリデータ
    """
    errors=(u'正しい値を入力してください。',)

    def __init__(self, pat):
        self.regex_pat=re.compile(pat)

    def validate(self, value):
        if not self.regex_pat.search(value):
            raise ValidationError(self.errors[0])
        return value
                    ⋮
```

次に、この正規表現バリデータを継承してURLとメールアドレスをチェックするバリデータを定義してみましょう。バリデーションチェックを行うvalidate()メソッドは、RegexValidatorに定義したものが共通して利用できるはずです。継承先のクラスではvalidate()メソッドは定義せず、親クラスに定義されているメソッドをそのまま利用することにします。

新たに定義する必要があるのは初期化メソッドのみです。URL、メールアドレスとも、形式が決まっていますから、初期化メソッドでは引数を受け取る必要がありません。初期化メソッドでは、URL、メールアドレスを判別するための正規表現文字列を決めうちで埋め込みます。

List06 バリデータクラス③（validators.py）

```
                    ⋮
class URLValidator(RegexValidator):
    """
    URLとして正しい文字列かどうかを調べるバリデータ
    """
    errors=(u'正しいURLを入力してください。',)

    def __init__(self):
        self.regex_pat=re.compile(
            r'^(http|https)://[a-z0-9][a-z0-9¥-¥._]*¥.[a-z]+'
            r'(?:[0-9]+)?(?:/.*)?$', re.I)


class EmailValidator(RegexValidator):
    """
    メールアドレスとして正しい文字列かどうかを調べるバリデータ
    """
    errors=(u'正しいメールアドレスを入力してください。',)

    def __init__(self):
        self.regex_pat=re.compile(
            r'([0-9a-zA-Z_&.+-]+!)*[0-9a-zA-Z_&.+-]+@'
            r'(([0-9a-zA-Z]([0-9a-zA-Z-]*[0-9a-z-A-Z])?¥.)+'
            r'[a-zA-Z]{2,6}|([0-9]{1,3}¥.){3}[0-9]{1,3})$')
```

これでバリデータモジュールは完成で、上記のコードを1つにまとめて「validators.py」というファイル名でcgi-binフォルダに保存してください。バリデータを実際に利用したサンプルプログラムは、「ウィジェット」を解説した後で示します。

12-03 ウィジェットの利用

Webアプリケーションでは、ユーザからデータを受け取るためのユーザインターフェース（UI）としてフォームを使います。フォームの実体はHTML文字列です。HTMLには文法や形式がありますので、フォームにも決まった形式やパターンがあります。

本書の第1章のサンプルプログラムでは、フォームを構成するHTML文字列をプログラムの中に埋め込んでいました。入力に誤りがあったときなど、フォームに前回入力した値を埋め込んで表示する必要があったからです。そのような処理を実現するためには、フォームをプログラムで動的に生成しなければなりません。

テンプレートエンジンを使うことによって、フォームを構成するHTMLをプログラムの中に埋め込む必要がなくなりました。同時に、入力ミスがあった項目の近くにエラーを表示するなどして、より効果的なUIを作れるようになりました。テンプレートエンジンを使うようになって、プログラムはスッキリしました。その代わり、テンプレートに条件分岐やループなど複雑なロジックを書くようになり、テンプレートが複雑になってしまっています。

WebアプリケーションのUIとなるフォームに求められる機能はたいてい似ています。たとえば、既存の値があればあらかじめ埋め込んで表示する。エラーがあればフォームの要素の近くに表示する、といった機能が求められるでしょうか。また、フォームを表現するためのHTML文字列にもパターンがあります。このようなパターンをうまく抽象化できれば、フォーム自体を部品化して、再利用しやすく実装できるかも知れません。

ここでは、Webアプリケーションで利用するフォームを効率的に扱う方法について考えてみたいと思います。

WebアプリケーションとCRUDフォーム

データベースと連携するWebアプリケーションでは、データを新規登録したり、更新するためにフォームを利用します。登録、更新に加え、データの削除をフォームを使って行うこともあるでしょう。また、データの内容を確認するためにデータの内容を表示することもあるはずです。

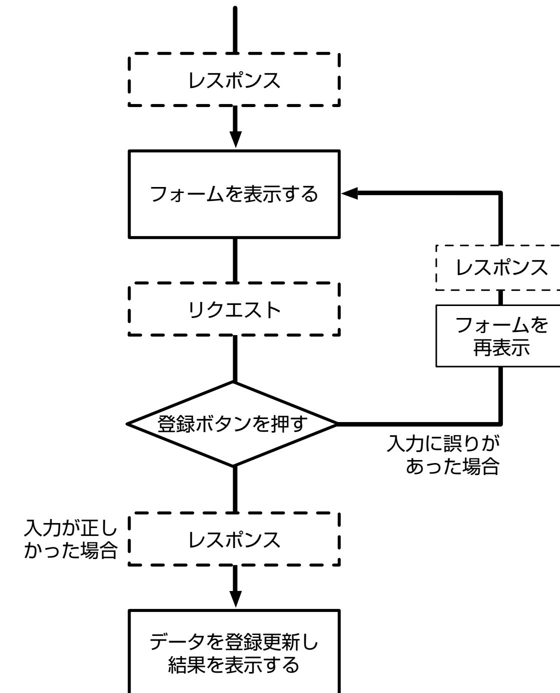
Webアプリケーションでデータベース上のデータを扱うときには、たいてい「新規登録」、「表示」、「更新」、「削除」が1セットになっています。このセットは**CRUD**（クラッド）と呼ばれています。**Create**（新規登録）、**Read**（読み込み/表示）、**Update**（更新）、**Delete**（削除）それぞれの頭文字を取った略語で、Webアプリケーションの開発でよく使われる用語です。

データの新規登録を行うフォームでは、どのような流れでデータの追加を行うべきかについて考えましょう。まずデータの入力に必要なフォームを、空の状態か、初期値が埋め込まれた状態で表示します。ユーザが項目を入力し、登録用のボタンを押すと、データがPOSTされます。Webアプリケーション側では、クエリとしてデータを受け取ります。

受け取ったデータをそのまま登録するわけにはいきません。Webのフォーム入力できる文字種などに制限がないので、もしかしたら不正な値が入力されているかもしれません。不正な値が登録されてしまうことを避けるため、バリデーションチェックを行う必要があります。

もし不正な値が見つかったら、入力に不備があることをユーザに分かりやすいように示しつつ、再度データを入力できるようにフォームを表示します。入力項目が複数あり、一部の入力値が正しかった場合には、以前に入力されていた値を再度表示します。正しいデータがPOSTされたら、データを登録します。

図01 新規登録の流れ



このように、Webブラウザから送るリクエストと、Webアプリケーションから送り返すレスポンスをうまく組み合わせて、正しいデータを登録できるような**遷移**を作ります。Webアプリケーションでは、リクエストとレスポンスは必ず**ペア**になっている必要があるということを思い出してください。リクエストを受け取り、Webアプリケーション側で適切なレスポンスを返す、ということを繰り返しながら、データ登録の流れを作ります。

次に登録済みのデータを更新するための流れについて考えてみましょう。新規登録と違い、あらかじめ登録されているデータをフォームで編集するわけですから、表示するときにデータが埋まった状態でフォームを表示します。その後、バリデーションチェックをする過程は「新規登録」の流れと同じです。必要に応じてエラーなどを表示しながら、正しいデータがPOSTされたらデータを更新します。

このように、典型的なWebアプリケーションでは、データの登録と更新に、よく似た仕組みを持つ必要があります。処理の流れだけでなく、表示するフォームの内容もとても良く似たものになるはずです。同じ種類のデータを対象にしているわけですから、バリデーションチェックの内容も同じになるはずです。

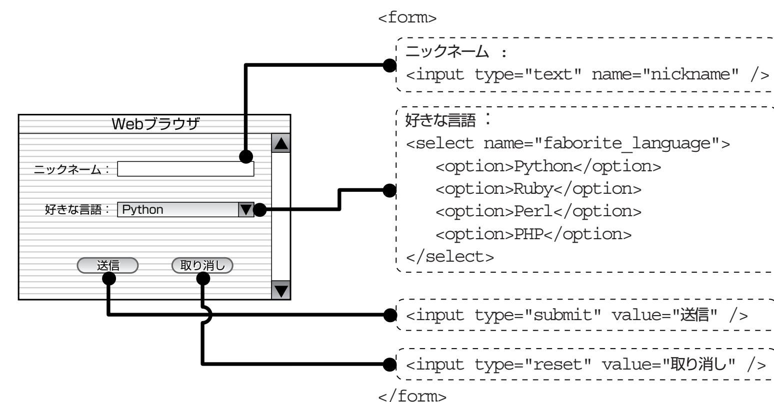
この2つの共通した部分の多いフォームをうまく扱えれば、Webアプリケーションで利用するフォームの処理を楽にできるかもしれません。

フォーム処理の抽象化

Webのフォームは、入力用の項目の集まりである、と定義できます。入力用の項目にはいくつかの種類があります。テキスト入力フィールド、メニューやチェックボックス、ボタンなどが主な種類です。入力用の項目はHTML文字列で表現します。入力項目の種類ごとに、HTML文字列の書き方が異なります。このようにして見ると、フォームというのは入力項目を部品としてまとめたもの、と捕らえることができます。

項目に入力された値に関しては、Webアプリケーション側でバリデーションチェックをかける必要があります。どのようなバリデーションチェックを実行すべきかについては、項目ごとに決まる場合が多いはずです。同じテキスト入力フィールドでも、場合によっては数値を入力することもありますし、メールアドレスを入力することもあります。フォームの部品にバリデータを設定できれば便利なはずです。

図02 フォームは入力項目をHTMLの部品としてまとめたもの



フォームの項目ごとに、項目を表現するためのHTMLを出力する仕組みを備えて、いろいろなバリデータを登録できるような仕組みがあると、フォームの処理をうまく抽象化できそうです。

ウィジェットとは

アプリケーションで利用するユーザインターフェース（UI）をプログラミングの部品として抽象化したものを**ウィジェット**（Widget）と呼びます。もともとは、GUIアプリケーションで利用するUIの部品を、クラスのような形で抽象化した仕組みを指してウィジェットと呼んでいました。「小物」という意味の英単語**gadget**と**window**を合成してwidgetと名付けられた、という説もあるようです。

Webアプリケーションで利用するウィジェットは、フォームの構成要素をクラスなどの形式で抽象化したものを指します。ウィジェットの実装方法に特に決まった形があるわけではありません。実装によって、フォームの表示のみに特化していたり、バリデータを含めて登録できるものなどさまざまな形態があるようです。ユーザインターフェースをプログラムで利用しやすいように抽象化したものである、という意味においては、Webアプリケーションのウィジェットは GUIアプリで利用されているものの流れをくんでいると言えるでしょう。

Webアプリケーションで利用されるウィジェットを簡単に説明すると、テンプレートの部品の一種、ということになります。Webアプリケーションの出力に使うテンプレートにフォームを、プログラムを使って自動的に生成するために利用するのがウィジェットです。そのようにすることによって、フォームに相当するHTMLをテンプレートに直接書く必要がなくなります。

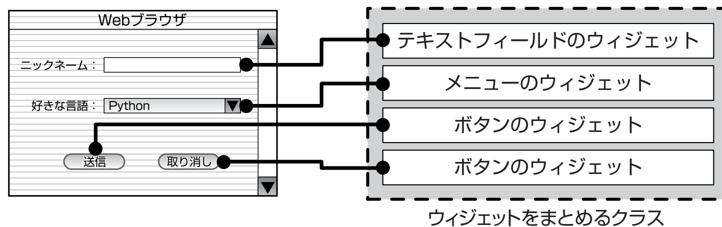
12-04 ウィジェットを作る

Webアプリケーションの開発をより効率的にするために、ウィジェットが有効である、ということが分かっていただけたでしょうか。ウィジェットに対する理解を深める意味もこめて、実際にウィジェットを作ってみることにしましょう。

Webアプリケーションで利用するウィジェットに求められる最小限度の要件は、フォームを部品として扱い、フォームを表示するためのHTML文字列を出力する、という機能です。今回作るウィジェットでは、HTMLを出力する機能に加えてバリデータも登録できるようにしましょう。ウィジェットを使って、フォームの表示とバリデーションチェックを行えるようにするわけです。

フォームの部品には、テキスト入力用のフィールドやラジオボタン、メニューなど、いろいろな種類があります。フォームの部品を使い分けられるように、部品の種類ごとにWidgetのクラスを定義しましょう。Webアプリケーションフォームに埋め込みたい部品ごとにWidgetのクラスインスタンスを作り、利用することになります。

図03 ウィジェットの機能



また、フォームの部品をひとまとめにして管理するために、Widgetのインスタンスを複数登録できるクラスを作ります。このクラスはWidgetのまとめ役となるクラスです。

ウィジェットの抽象クラスを作る

フォームの部品となるウィジェットのクラスを作る前に、ウィジェットクラスの親となる抽象クラス（BaseWidgetクラス）を作りましょう。

ウィジェットはフォームの部品となるHTML文字列を出力する役目を持っています。フォームの部品には、クエリのキーとなるnameアトリビュートの値やフォームの**ラベル**など、固有の情報を持たせる必要があります。この情報は状況によって変化します。このような情報は、ウィジェットの初期化メソッドに引数として渡すとよいでしょう。

以下のコードは、ウィジェットの抽象クラスの初期化メソッドの定義部分です。フォームの部品となるHTML文字列を作るために、先ほど作ったテンプレートエンジンを利用します。そのため、まずテンプレートエンジン（SimpleTemplate）をインポートしています。

List07 初期化メソッド（widgets.py）

```
from simpletemplate import SimpleTemplate

class BaseWidget(object):
    """
    ウィジェットのベースクラス
    """

    def __init__(self, name, label='', options=None,
                 validators=[], attrs={}):
        self.name=name
        self.label=label
        self.options=options
        self.validators=validators
        self.attrs=" ".join('%s="%s"'%(k, v)
                            for k, v in attrs.items())
                  ⋮
```

初期化メソッドでは、インスタンスオブジェクト自体を指す引数selfの他に、5つの引数を受け取っています。name以外の引数には初期値が割り当てられているので、オプションの引数となっています。**name**はフォームのnameエレメントに埋め込む文字列を渡します。**label**という引数には、フォームの部品の名称を示すラベルに表示する文字列を渡します。

optionsにはシーケンスを渡します。メニューやラジオボタンなど、複数の項目から1つを選ぶようなフォームの部品で利用することを目的とした引数です。

validatorsには、フォームの部品で利用するバリデータのインスタンスを渡します。フォームのクラスはバリデーションチェックを行うメソッドを持っていて、初期化メソッドに渡されたバリデータのインスタンスを使ってバリデーションチェックを行います。

atrrsという引数は、フォームのエレメントに埋め込むアトリビュートを指定するための引数です。フォームの部品となるエレメントには、nameアトリビュートだけでなくいろいろなアトリビュートを指定することがあります。たとえばフォームの見栄えを変更する目的でstyleアトリビュートやclassアトリビュートを設定することがあります。この引数は、そのような細かなカスタマイズを行うために用意しています。引数には辞書を渡し、辞書のキーと値でそれぞれアトリビュート名とアトリビュート値を指定します。

ウィジェットのクラスでは、抽象クラスで定義した初期化メソッドを共通して利用しています。

次に、フォームのHTML文字列を作るメソッドを定義します。テンプレートエンジンを使ってフォームの部品を表示します。

List08 HTMLを組み立てるメソッド群(widgets.py)

```
                    :
    def get_label(self, error):
        body=("""<label for="${name}">${label}¥n"""
              """$if error:¥n"""
              """<span class="error">${error}</span>¥n"""
              """$endif¥n"""
              """</label>""")
        t=SimpleTemplate(body)
        return t.render({'name':self.name, 'label':self.label,
                         'error':error})

    def get_form(self, value=None):
        return ''

    def display(self, value=None, error=None):
        return self.get_label(error) + self.get_form(value)
                    :
```

このウィジェットでは、ラベルとフォーム本体を分けて表示します。ラベルの部分はどのフォーム部品でも共通化できますので、抽象クラスに共用するget_label()というメソッドを定義しています。

get_label()メソッドの内部では、テンプレートエンジンに渡すテンプレート文字列を文字列リテラルとして埋め込んでいます。ラベル部分には、メソッドの引数として渡したエラー文字列を表示するようにします。

get_form()メソッドはフォーム部品の本体となるHTML文字列を表示するためのメソッドです。このメソッドの返す文字列はウィジェットの種類によって異なります。抽象クラスでは、共通して利用するメソッド名を決めるために空の文字列を返す単純なメソッドを定義しています。また、このメソッドにはvalueという引数を渡します。フォームの部品にあらかじめ埋め込んでおきたい文字列などを渡すための引数です。

display()メソッドはウィジェットの出力するHTML文字列全体を返すためのメソッドです。ウィジェットを使う側では、このメソッドを呼び出してHTML文字列を取得します。このメソッドはget_label()とget_form()を間接的に呼び出し、ウィジェットの文字列全体を組み立てて返します。それぞれのメソッドに渡すために、引数valueとerrorを受け取っています。

最後に、バリデーションチェックを行うためのメソッドを定義します。初期化メソッドに引数として渡されたバリデータを元にバリデーションチェックを行います。

List09 バリデートメソッド(widgets.py)

```
                    :
    def validate(self, value):
        from validators import ValidationError
        error=None
        for v in self.validators:
            try:
```

```
                value=v.validate(value)
            except ValidationError, e:
                error=e.msg
        return value, error
                         ⋮
```

バリデータはシーケンスに複数指定されて渡されることがあります。**必須項目で、かつ整数**というような複合的なバリデーションチェックに対応するためです。ループを組んでバリデータを取り出しつつ、validate()メソッドを使ってバリデーションチェックをかけています。

バリデーションチェックで値が不正であることを見つけると、ValidationErrorという例外が投げられます。バリデーションチェック時には、ValidationErrorという例外だけをexcept文で捕まえるようになっています。例外オブジェクトのmsgアトリビュートにはチェックが失敗した理由が文字列で格納されています。この文字列を取り出し、変数に代入しています。

このメソッドは2つの戻り値を持っています。1つは、バリデータによって変換された入力値です。もう1つはエラーです。2つ目の引数は、エラーが起こったときにのみ文字列が代入されます。エラーが起こらなかった場合にはNoneが返ります。

テキスト入力フォーム用のウィジェットを作る

ウィジェットの抽象クラスが定義できましたので、次はフォームの部品に対応するウィジェットのクラスを定義しましょう。まずは、文字列を入力するために利用する2種類のウィジェットを定義します。

初期化メソッドやバリデーションチェック用のメソッドは抽象クラスに定義してあります。ウィジェットに定義する必要があるのは、フォームの本体に相当するHTML文字列を出力するためのget_form()メソッドです。以下がテキストフィールド、テキストエリア用クラスの定義になります。

List10 テキスト入力フォーム用のウィジェット(widgets.py)

```
                         ⋮
class Text(BaseWidget):
    """
    テキストフィールド用のウィジェット
    """

    def get_form(self, value=''):
        body=("""<input type="text" name="${name}"
                        value="${value}" """
              """ ${attrs} />""")
        t=SimpleTemplate(body)
        return t.render({'name':self.name, 'value':value,
                         'attrs':self.attrs})


class TextArea(BaseWidget):
    """
    テキストエリア用のウィジェット
    """
    def get_form(self, value=''):
        body="""<textarea name="${name}"
${attrs}>${value}</textarea>"""
        t=SimpleTemplate(body)
        return t.render({'name':self.name, 'value':value,
                         'attrs':self.attrs})
                         ⋮
```

どちらのクラスも、get_form()メソッドを定義していて、テンプレートエンジンを使ってフォーム本体を表現するためのHTMLを出力しています。HTMLの中には、初期化メソッドで受け取った文字列などを埋め込んでいます。また、既定値をフォームに埋め込む用途を考慮して、valueという引数をフォームに埋め込んで表示するようにテンプレートを書いてあります。

なお、テンプレートの文字列を埋め込んであるリテラルの部分には、見慣れない表記が見えます。Pythonは、複数の文字列リテラルを丸カッコで囲むと、それぞれの文字列を連結して扱ってくれるのです。複数行にわたる長い文字列をスマートにソースに埋め込みたいときに利用すると便利なテクニックです。

メニュー・ラジオボタン用のウィジェットを作る

では次に、フォームに表示するメニュー(selectエレメント)を作るためのウィジェットクラスを定義してみましょう。get_form()メソッドのみを定義

する、ということについてはテキストフィールドなどと変わりませんが、フォーム部品を表示するためのHTMLを出力テンプレートが少し複雑になっています。

List11 メニューウィジェット(widgets.py)

```
                    ︙
class Select(BaseWidget):
    """
    メニュー用のウィジェット
    """
    def get_form(self, value=''):
        body=("""<select name="${name}" ${attrs}>¥n"""
              """$for v in options:¥n"""
              """<option value="${v[0]}"¥n"""
              """$if value==v[0]:¥n"""
              """ selected ¥n"""
              """$endif¥n"""
              """>${v[1]}</option>¥n"""
              """$endfor¥n"""
              """</select>¥n""")
        t=SimpleTemplate(body)
        return t.render({'name':self.name, 'value':value,
                         'options':self.options,
                         'attrs':self.attrs})
                    ︙
```

メニュー用のウィジェットでは、メニューに表示する項目を複数HTMLの中に埋め込む必要があります。メニューに表示する項目は、文字列のシーケンスの形でウィジェットの初期化メソッドに引数として渡します。テンプレートでは、引数の内容を保存したアトリビュートを使って、項目を展開しています。

テンプレートの中には**$for〜:**構文を使ったループが見えます。ここで、初期化メソッドに渡されたシーケンスを展開しています。フォームのメニュー項目では、<option>タグを使ってメニューの項目を表示します。ループを使って、必要な数だけoptionエレメントを繰り返しています。

メソッドに引数valueが渡されていた場合は、valueと同じ文字列を持つ項目を選択状態で表示します。メニューを選択状態で表示するにはselectedというアトリビュートをエレメントの中に記入します。そのために、条件分岐のロジックがループの中に埋め込まれています。

List12 ラジオボタンウィジェット(widgets.py)

```
                    ︙
class Radio(BaseWidget):
    """
    ラジオボタン用のウィジェット
    """
    def get_form(self, value=''):
        body=("""$for v in options:¥n"""
              """${v[1]} : """
              """<input type="radio" name="${name}" value=
              "${v[0]}"¥n"""
              """$if value==v[0]:¥n"""
              """ checked ¥n"""
              """$endif¥n"""
              """>¥n"""
              """$endfor¥n""")
        t=SimpleTemplate(body)
        return t.render({'name':self.name, 'value':value,
                         'options':self.options,
                         'attrs':self.attrs})
                    ︙
```

ラジオボタン用のウィジェットでも、同様に複数の項目を繰り返す必要があります。テンプレートの繰り返しロジックを使って、複数のエレメントを文字列として生成しています。

ラジオボタンの場合は、同じ値を持つnameアトリビュートを埋め込んだinputエレメントを複数並べる形になります。テンプレート全体をループか囲む形になっているのはそのためです。

メニューと同じように、value引数が指定されたときのために条件分岐のロジックが埋め込まれています。ラジオボタンの場合には、**checked**というアトリビュートを埋め込むことで選択状態の表示になります。

サブミットボタンとリセットボタン

サブミットボタンとリセットボタン用のウィジェットは、次のとおりです。それぞれ所定のinputエレメントを出力しているだけですので、特に解説する必要はないでしょう。

List13 ボタン類のウィジェット(widgets.py)

```
                        ⋮
class Submit(BaseWidget):
    """
    サブミットボタン用のウィジェット
    """
    def get_label(self, error):
        return ''

    def get_form(self, value=''):
        body=("""<input type="submit" value="${label}" """
              """ ${attrs} />""")
        t=SimpleTemplate(body)
        return t.render({'label':self.label, 'attrs':self.attrs})

class Reset(Submit):
    """
    リセットボタン用のウィジェット
    """
    def get_form(self, value=''):
        body=("""<input type="reset" value="${label}" """
              """ ${attrs} />""")
        t=SimpleTemplate(body)
        return t.render({'label':self.label, 'attrs':self.attrs})
                        ⋮
```

フォームをまとめるウィジェットを作る

フォームの部品となるウィジェットを一通り作りました。最後に、ウィジェットをまとめて登録し、フォームとして管理するためのウィジェットを作りましょう。ウィジェットクラスのインスタンスを複数シーケンスとして登録して利用します。フォーム全体を表現するためのHTML文字列を出力する機能と、フォーム全体のバリデーションチェックを行う機能も持たせます。

まずは、クラスの定義と初期化関数を定義します。このクラスインスタンスは、フォーム部品となるウィジェットをシーケンスとして保持します。また、formエレメントのactionアトリビュートやmethodアトリビュートなどをこまかくコントロールするためにアトリビュートも指定できるようにします。

ウィジェットのシーケンスは初期化メソッドに引数として渡します。アトリビュートを指定するためには、ウィジェットのクラスと同様に辞書を引数として渡すようにしましょう。

List14 Formクラスの初期化メソッド(widgets.py)

```
                        ⋮
class Form(object):
    """
    ウィジェットを登録するフォーム用クラス
    """
    def __init__(self, forms, attrs={}):
        self.forms=forms
        self.attrs=" ".join('%s="%s"'%(k, v)
                            for k, v in attrs.items())
                        ⋮
```

次に、クラスインスタンスに登録したフォーム全体を表示するためのメソッドを定義します。初期化メソッドで登録したウィジェットのシーケンスを使ってループを組みます。それぞれのウィジェットのHTML文字列を生成し、全体をformエレメントで囲んで表示します。

このメソッドは、フォームに埋め込んで表示する既定値と、ウィジェットに表示するエラーを辞書として引数に受け取ります。それぞれのウィジェットの既定値とエラー文字列は、フォームのnameをキーとして辞書に埋め込まれています。ループを処理する過程で、引数の辞書から既定値と値を取り出し、ウィジェットに渡しています。

List15 フォームの表示メソッド(widgets.py)

```
                        ⋮
    def display(self, values={}, errors={}):
        container=''
        for f in self.forms:
            container+=f.display(values.get(f.name, ''),
                                 errors.get(f.name, ''))
            container+="""<br clear="all" />"""
        body=("""<form ${attrs}>¥n"""
              """${container}¥n"""
              """</form>¥n""")
        t=SimpleTemplate(body)
        return t.render({'attrs':self.attrs, 'container':
        container})
                        ⋮
```

最後はフォームに登録されたウィジェット全体のバリデーションチェックを行うためのメソッドを定義します。アトリビュートからウィジェットを取り出し、それぞれのウィジェットに対してバリデーションチェックをかけていきます。

ウィジェットのバリデーションチェック用メソッドは、バリデータが変換した値とエラーを戻り値として返します。その結果を辞書に登録して、値用の辞書、エラー用の辞書2つを返り値として返します。

List16 Formクラスのバリデートメソッド (widgets.py)

```
                        ⋮
    def validate(self, invalues):
        errors={}
        values={}
        for f in self.forms:
            value, error=f.validate(invalues.get(f.name, ''))
            values[f.name]=value or ''
            if error:
                errors[f.name]=error
        return values, errors
```

これでwidgetsモジュールは完成です。上記のコードを1つにして「widgets.py」というファイル名でcgi-binフォルダに保存してください。

ウィジェットとバリデータを使ったサンプルプログラム

ウィジェットとバリデータを使ったサンプルプログラムを作ってみましょう。これまでもフォームや入力値のチェック（バリデーションチェック）を行うプログラムをいくつか作ってきました。今回はウィジェットとバリデータを使って、項目の多いフォームを作ってみることにしましょう。

入力項目の多いWebアプリケーションとして思いつくのが**アンケートフォーム**です。アンケートフォームとは、Webブラウザに表示したフォームに項目を入力し、アンケートを収集するためのWebアプリケーションのことです。このようなWebアプリケーションを作るときには、必須項目の記入漏れや打ち間違いのチェックが欠かせません。チェックなしに入力値を受け取ってしまうと、有効なデータが集められないのです。

プログラムでリクエストを受け取り、1つ1つの項目について値をチェックすることでもアンケートを作ることはできます。しかし、項目が多くなるとチェックが大変になり、プログラムの不具合も出やすくなります。また、フォームの数が多くなるとフォームを作るのも大変になります。

そこでここでは、ウィジェットとバリデータを活用して、気の利いたバリデーションチェック処理を含んだWebアプリケーションを作ってみます。フォームの項目管理と表示にウィジェットを使い、入力値のチェックにはバリデータを使います。また、データはO/Rマッパーを経由してデータベースに登録するようにします。

まず、Webアプリケーションで使うO/Rマッパーのクラスやウィジェットを定義するところから始めましょう。この定義は、Webアプリケーション本体のプログラムとは別の「widgettest_classes.py」というファイルに保存するようにします。

以下がO/Rマッパーのクラスを定義する部分です。POSTされた各項目について、データベースに保存できるようなカラムを定義しています。その後、必要があればテーブルを作っています。

List17 O/Rマッピング処理 (widgettest_classes.py)

```
from os import path
import sqlite3
from simplemapper import BaseMapper

class Profile(BaseMapper):
    rows=(('lastname', 'text'), ('firstname', 'text'),
          ('birthyear', 'int'), ('gender', 'int'),
          ('email', 'text'), ('url', 'text'),
          ('language1', 'text'), ('language2', 'text'),
          ('comment', 'text'))

p=path.join(path.dirname(__file__), 'questionnaire.dat')
con=sqlite3.connect(p)
BaseMapper.setconnection(con)
Profile.createtable(ignore_error=True)
                        ⋮
```

次に、フォームを管理するためのウィジェットを定義します。先ほど作ったウィジェットでは、フォームの部品となるウィジェットをクラスとして定義し、ウィジェットをまとめるFormクラスに引数として渡す形式でフォー

ムを定義することになっていました。また、それぞれのウィジェットクラスでは、クラスインスタンスを作るときに、フォームのラベルや項目の他、バリデータを指定できるようになっています。

List18 フォームの管理 (widgettest_classes.py)

```
                      ⋮
from validators import NotEmpty, IntValidator, IntRangeValidator,¥
                      URLValidator, EmailValidator,
ValidationError
from widgets import Text, Select, Radio, Submit, Reset, Form

languages=[('', '---')]+[(x, x) for x in ['Perl', 'PHP',
          'Python', 'Ruby']]
forms=( Text('lastname', u'名字', validators=(NotEmpty(),)),
        Text('firstname', u'名前', validators=(NotEmpty(),)),
        Select('birthyear', u'生まれた年',
            options=[('0', '---')]+¥
                 [(str(x), str(x)) for x in range(1940, 2007)],
            validators=(NotEmpty(),
            IntRangeValidator(1900, 2007),)),
        Radio('gender', u'性別',
            options=(('1', u'男性'), ('2', u'女性')),
            validators=(IntRangeValidator(1, 2),)),
        Text('email', u'メールアドレス',
            validators=(EmailValidator(),), attrs={'size':'40'}),
        Text('url', u'URL',
            validators=(URLValidator(),), attrs={'size':'40'} ),
        Select('language1', u'一番好きな言語は?',
            options=languages, validators=(NotEmpty(),)),
        Select('language2', u'二番目に好きな言語は?',
            options=languages, validators=(NotEmpty(),)),
        Text('comment', u'一言', attrs={'size':'40'} ),
        Submit('submit', u'登録'), Reset('reset', u'クリア'))
form=Form( forms, {'action':'widgettest.py', 'method':'POST'} )
```

上記のサンプルでは、ウィジェットのタプルをいったんformsという変数に定義しています。タプルの中には、ウィジェットのクラスインスタンスの定義が並んでいます。この例では、変数などを経由せずクラスを直接定義して、オブジェクトとしてタプルに代入しています。

フォームに表示したい項目の種類によって扱うウィジェットのクラスを分けて定義しています。表示したい項目に合わせて必要なクラスを定義します。

また、表示に必要な項目、バリデータなどはクラスの初期化時に引数として渡しています。括弧の対応をよく見ながら、クラス定義の区切りに注意してコードを見ると内容をよく理解できるはずです。

formsという変数に代入したウィジェットのタプルは、Form () クラスのインスタンスを作る際に引数として渡しています。

テンプレートの作成

次に、フォームを表示するためのテンプレートを作ります。フォームの表示はウィジェットが担当するので、テンプレートはとてもシンプルになっています。テンプレートではフォームオブジェクトなどを受け取り、必要に応じて表示します。また、データの登録がうまくいったときにも同じテンプレートが使えるよう、**$if**構文を使って簡単なロジックが埋め込んであります。

List19 questionform.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
<link rel="stylesheet" href="/style.css" type="text/css"/>
<style type="text/css"><!--
    label {display:block; font-weight: bold;}
    .error {color: #b21; font-weight: normal;}
-->
</style>
</head>
<body>
    <h1 class="header">アンケート</h1>
    $if not dataposted:
        ${form.display(values, errors)}
    $endif
    $if dataposted:
        <p>アンケートを登録しました</p>
    $endif
</body>
</html>
```

Webアプリケーションの作成

最後に、Webアプリケーション本体となるプログラムを作ります。O/Rマッパーのクラス、ウィジェットやテンプレートエンジンを活用して、処理を

行います。30行弱のとても短いプログラムです。フォームの表示、およびクエリを受け取る処理を1つのプログラムで担当しています。

List20 widgettest.py

```
#!/usr/bin/env python
# coding: utf-8

from simpletemplate import SimpleTemplate
from os import path
from httphandler import Request, Response
from widgettest_classes import Profile, form
import cgitb; cgitb.enable()

req=Request()
values={}
[values.update({k:req.form.getvalue(k, '')})
                    for k in req.form.keys()]
cvalues, errors=form.validate(values)
if len(req.form.keys())==0:
    errors={'foo':'bar'}

res=Response()
p=path.join(path.dirname(__file__), 'questionform.html')
t=SimpleTemplate(file_path=p)

post_values={'form':form, 'values':values, 'errors':errors,
             'dataposted':False}
if not errors:
    post_values.update(dataposted=True)
body=t.render(post_values)

res.set_body(body)
print res
```

このプログラムで実行していることはとても単純です。

- フォームを表示し
- リクエストを受け取り
- バリデーションチェックをかけて
- 結果を表示する

ということだけです。バリデーションチェックの結果、入力に誤りがあったり、期待通りの値が入力されなかった場合には、フォームを再表示し、誤りの原因を表示します。

図04 エラーが起こったら、フォームにエラーを表示する

プログラムの流れはとてもストレートですが、ユーザになにをすべきかを的確に表示し、正しいアンケートを収集するための十分な仕組みが備わっています。また、プログラムの基本的な流れと、フォーム項目などの設定部分が綺麗に分離しているのもこのWebアプリケーションの特徴です。プログラムの流れ（遷移）を変更したいときには本体のプログラムを修正することになります。また、フォームの項目を増やしたり、バリデーションチェックの項目を変更するときにはウィジェットの定義を変更します。

Webアプリケーションの処理を抽象化することによって、プログラミングの効率がとても高くなります。同時に、プログラム内部での分化が進んで、プログラム全体の見通しがよくなるわけです。

CHAPTER 13

Webアプリケーションサーバを使った開発

テンプレートエンジン、O/Rマッパーなどを使うことで、Webアプリケーションの開発はかなり効率的になりました。ここでは、もう一歩踏み込んで、Webアプリケーションの開発をさらに効率化するための方法について検討してみましょう。

13-01　URLと遷移

RSSリーダーに巡回用URLを登録するフォームを追加したときのことを思い出してください。巡回用URLを登録する仕組みを作るために、3つのプログラムを作りました。1つは登録済みのURLをリスト表示するプログラム。そして登録済みのURLを編集するためのプログラム。もう1つは、新規にURLを登録するためのプログラム。巡回用URLを管理するのに必要な3つの機能を実現するために、3つのプログラムを作りました。

Webアプリケーションでは、リクエストとレスポンスは必ずペアになっている必要があります。つまり、ユーザに画面を見せるためには、Webサーバに対してリクエストを送る必要があるわけです。

リクエストは**URL**と**クエリ**から構成されています。たいていのWebアプリケーションでは、次の画面に移動するために、別のURLに**遷移**します。別のURLを変えることで、リクエストをWebサーバに送信し、新しい画面を表示するわけです。

本書で開発に使っているWebサーバでは、プログラムのファイル名がURLの一部になります。そのため、クエリ部分を除いたURLを変えるためには、別のプログラムファイルを作る必要があります。

CGIとURL

このようにしてWebサーバ上でプログラムを動かすための仕組みを**CGI**と呼びます。CGIは「Common Gateway Interface」の頭文字を取った略称です。本来はWebサーバとプログラムの間でデータをやりとりするための仕組みを定めた仕様のことをCGIと呼びます。Webサーバを動かすプログラムを指してCGIと呼ぶこともあるようです。

CGIは次のような仕組みで動きます。ちょうど、WebサーバがURL上のパスをフォルダの階層やファイル名と解釈して、HTMLや画像ファイルをWebブラウザなどに返すのに似た働きをします。

- **URLのパスの一部をファイル名として解釈して、呼び出すプログラムを探す**
- **クエリを環境変数に格納してプログラムを起動する**
- **プログラムから返ってきた結果をレスポンスとして返す**

本書でこれまで開発に利用してきたWebサーバでも、URLの一部をプログラムのファイル名として解釈してプログラムを動かします。このため、個別のURLを設けて表示する画面やフォームを変えようとすると、それぞれのURLに対応するプログラムファイルを用意する必要があります。結果として、多くのプログラムファイルを作る必要があるわけです。

しかし、一般的なプログラムの開発では、1つの処理を実行するためだけに個別にプログラムファイルを作る、ということはあまり行いません。個別の処理は関数やメソッドなどとして実装して、関連する処理をモジュールやクラスといった形でまとめる、といった手法を使うのが普通です。

今我々が利用している開発手法では、特定の処理を実行するために1つのプログラムファイルを作る必要があります。これはちょうど、1つの関数やメソッドを作るために1つのプログラムファイルを作る、というような話です。また、テンプレートエンジンやO/Rマッパーを使うようになって、簡潔で短いプログラムを書けるようになりました。短いプログラムが、多くのプログラムファイルに分散してしまうのでは、プログラムの編集がしづらくなってしまいます。

1つのファイルに1つの処理、という形式をとらずに、モジュールやクラスを定義するのと同じようにWebアプリケーションを開発できないものでしょ

うか。そうすれば、より効率的に、高機能なWebアプリケーションを作れるようになるはずです。

13-02 Webサーバの動く仕組みを理解する

さて、ここで少し趣向を変えて、Webサーバの動く仕組みについて簡単に見ていきましょう。Webサーバの基本的な動作は、**リクエストを受けてレスポンスを返す**という単純なものです。Webサーバ自体のプログラムもそれほど難しくはありません。

これまで開発に使っていたCGIHTTPServerというWebサーバは、すべてPythonで書かれています。このWebサーバのソースコードを見れば、Webサーバの動く仕組みを理解できるはずです。

CGIHTTPServerは、大きく分けて2つのクラスから作られています。1つは、ネットワークの通信を担当するHTTPServerというクラスで、このクラスはBaseHTTPServerというモジュールに定義されています。もう1つは、リクエストを受け取り実際に処理を行うためのクラスです。CGIHTTPServerでは、CGIを実行するために必要な部分のみがCGIHTTPRequestHandlerというクラスに定義されています。このクラスはSimpleHTTPRequestHandlerというクラスを継承していて、静的なファイルを配信する一般的なWebサーバに求められる機能はこのクラスに定義されています。SimpleHTTPRequestHandlerはBaseHTTPRequestHandlerという抽象クラスを継承しています。

CGIHTTPRequestHandlerやSimpleHTTPRequestHandlerといったクラスは、俗に**ハンドラ**と呼ばれています。リクエストを処理（ハンドリング）するための方法がハンドラとなるクラスの中に抽象化されています。必要に応じてクラスを拡張することで、リクエストをいろいろな方法で処理できます。

ファイルをレスポンスとして送信する

ハンドラクラスでは、リクエストに応答するためのハンドラメソッドが定義されています。たとえば、SimpleHTTPRequestHandlerにはGETリクエストを処理するためのdo_GET()メソッドが定義されています。send_head()というメソッドを呼び出し、戻り値としてファイルオブジェクトを受け取ってレスポンスとして返す、というとても簡潔なメソッドとなっています。self.wfileというオブジェクトが、リクエストとして返すファイルオブジェクトとして扱われているわけです。

List01 do_GET()メソッドの実装（SimpleHTTPServer.py）

```
                    ⋮
    def do_GET(self):
        """Serve a GET request."""
        f = self.send_head()
        if f:
            self.copyfile(f, self.wfile)
            f.close()
                    ⋮
```

さて、send_head()メソッドではどのような処理を実行しているのでしょうか。send_head()メソッドは、ドキュメンテーション文字列を取り除くと30行弱の短いコードでできています。

List02 send_head()メソッドの実装（SimpleHTTPServer.py）

```
                    ⋮
    def send_head(self):
        path = self.translate_path(self.path)
        f = None
        if os.path.isdir(path):
            for index in "index.html", "index.htm":
                index = os.path.join(path, index)
                if os.path.exists(index):
                    path = index
                    break
            else:
                return self.list_directory(path)
        ctype = self.guess_type(path)
        if ctype.startswith('text/'):                    ❶
            mode = 'r'
        else:
            mode = 'rb'
        try:
            f = open(path, mode)
        except IOError:
            self.send_error(404, "File not found")
```

```
        self.send_response(200)
        self.send_header("Content-type", ctype)                    ❷
        fs = os.fstat(f.fileno())
        self.send_header("Content-Length", str(fs[6]))
        self.send_header("Last-Modified",
                         self.date_time_string(fs.st_mtime))
        self.end_headers()
        return f
```

このメソッドでは、まず最初にURLをファイル上のパスに変換しています。その後、もしパスがフォルダやディレクトリだったら、インデックスページを表示するか、あるいはディレクトリにあるファイル一覧を表示する、という場合分けをしています。

その後、ファイルの拡張子などからファイル種類を判別しています。レスポンスの種類を判別するためにヘッダに記載する情報を、ここで調べているわけです（❶）。

その後には、ファイルを読み込むために開くコードが続いています。ファイルが開けなかったらエラーのステータスを返します。ファイルが開けたら、正常な動作であることを示す200という番号をステータスとして返します。このステータスは**ステータス行**と呼ばれる文字列に変換してWebブラウザなどに送信します。

その後は、ヘッダ行を複数返しています（❷）。send_header()はステータス行とヘッダを送るためのメソッドなので、本文は送信しません。do_GET()メソッドがsend_header()で開いたファイルを戻り値として受け取り、ファイルの内容をレスポンス本文として返しています。

このように、ネットワーク通信以外のWebサーバの処理は簡単に記述できるのです。

Webサーバ内でプログラムを起動する

CGIHTTPRequestHandlerには、do_POST()というPOSTリクエストを処理するためのメソッドが定義されています。このメソッドで、ファイルとして置かれたプログラムを起動し、レスポンスとして返しています。do_POST()メソッドが、これまで使っていたWebサーバでプログラムを起動していた本体ということになります。

do_POST()メソッドはdo_GET()メソッドと同様に、10行に満たないとても短いプログラムです。実際にプログラムを起動しているのはrun_cgi()という200行ほどのメソッドです。ファイルを読み込むだけのsend_header()メソッドに比べてプログラムの行数が長くなっているのにはいくつか理由があります。CGIプログラムにいろいろな情報を受け渡すために多くの環境変数を設定していることが1つの理由です。たとえば、Webブラウザなどから送られてくるクエリを取り出し、環境変数に渡す必要があります。たとえば、run_cgi()メソッドの冒頭では、URLを**?**で分割してクエリ文字列を取り出しています。

List03 run_cgi()メソッドの一部（CGIHTTPServer.py）

```
    def run_cgi(self):
        """Execute a CGI script."""
        dir, rest = self.cgi_info
        i = rest.rfind('?')
        if i >= 0:
            rest, query = rest[:i], rest[i+1:]
        else:
            query = ''
                      ⋮
```

クエリはURL上に乗って送られてくる場合もありますし、リクエスト本文に記載されている場合もあります。いろいろな可能性を考慮して、コードを書く必要があるわけです。

また、Webサーバ内でプログラムを起動し、標準出力から結果を取り出すためにサブプロセスを呼び出しています。プロセス管理はWindowsやLinuxなどOSによって処理の方法が異なります。いろいろな環境でWebサーバを動かすために、プログラム内部で場合分けをしているためにプログラムが長くなっています。

Webサーバの基本的な働き

さて、このようにしてみると、Webアプリケーションはとても単純な仕組みで動いていることが分かります。

● **URLを解釈する**
リクエストとして返すファイルなどを特定するために、URLをディレクトリ構造として分割して目的のファイルを見つけ出します。

● **ステータス行を送る**
これから送ろうとするレスポンスがどのようなものかを文字列で簡潔に示すためにステータス行と呼ばれる文字列を送信します。

● **ヘッダを送る**
レスポンスの内容について、付加的な情報をWebブラウザなどに伝えるために、ヘッダ行を送ります。

● **レスポンス本文を送る**
Webブラウザなどに対してレスポンスの本文を送ります。

一般的なWebサーバは、URLの一部をフォルダやファイルの構造（パス）として解釈して、見つけたファイルをそのままレスポンスとして送り出します。URLの一部をパスとして解釈して見つけたファイルがプログラムだった場合には、プログラムを実行して結果をレスポンスとして返します。このように動作するようにコードが書いてあるからこのような処理をしているだけであって、すべてのWebサーバがこのように動くべき、ということはまったくありません。ハンドラメソッドを変更することで、まったく異なったWebサーバを作ることも当然できます。

13-03　Webアプリケーションサーバを作る

Pythonを使った一般的な開発と同じように、Webアプリケーションを開発する手法があれば、開発はより効率的になるはずです。ここで言う**一般的な開発手法**とは、1つの処理を1つごとのファイルに分けるのではなく、個別の処理は関数などとして実装して、関連する処理をモジュールのような形にまとめる、といった手法のことです。

これまで見てきたように、表面に見えるWebサーバの機能や働きは比較的単純です。たとえば、パスをフォルダやファイルの構造として解釈するのではなく、関数名として解釈するWebサーバも作れるはずです。URLを関数名として解釈して呼び出し、関数の返り値をレスポンスとして返します。このようなWebサーバができれば、個別の処理を関数として複数プログラムに定義できるようになり、いちいち個別のファイルに分けずに済むようになるはずです。

このように、Webアプリケーションの開発に特化したWebサーバのことを**Webアプリケーションサーバ**と呼ぶことがあります。HTMLや画像などの静的なファイルをレスポンスとして返すためのWebサーバと区別する目的でWebアプリケーションサーバという言葉がよく使われます。

Webアプリケーションサーバの仕様を決める

Webアプリケーションサーバを作る前に、簡単に仕様を決めましょう。リクエストを受け取ったら関数を呼び出し、関数が返した結果をレスポンスとして返す、というのが最低限求められる要件です。

ただし、すべての関数をリクエストから呼び出せるのでは、セキュリティ上の問題が発生するかもしれません。内部利用を目的に作った関数まで不意に呼び出されてしまうかもしれないからです。そこで、Webアプリケーションサーバにあらかじめ登録した関数だけを呼び出せるようにしましょう。また、リクエストとして送られてくるクエリは、関数の引数として受け取れるようにしましょう。

リクエストを受け取り、関数に渡すには、GETとPOSTリクエストを処理しているハンドラメソッドを書き換えればよいはずです。メソッドの中でリクエストを解釈し、クエリを分割します。分割したクエリは、呼び出し用に登録された関数に引数として渡します。

Webアプリケーションサーバから呼び出された関数は、内部で必要な処理を行いレスポンスとして返します。関数は、レスポンス本文だけでなく、ヘッダを含む完全なレスポンスを返す必要があります。以前作ったテンプレートエンジン（SimpleTemplate）とResponseクラスを活用することにしましょう。

Webアプリケーションで画像やCSSなどの静的なファイルを扱いたいこと

があるかもしれません。そのような場合に対応できるように、/static/～のように特定のパスから始まるリクエストを受けたときには静的なファイルをレスポンスとして返すようにしましょう。

クラス定義とリクエストを受け取るメソッド

Webアプリケーションサーバを作るに当たり、必要になるのは**ハンドラクラス**を定義することです。ハンドラクラスのベースとなるクラスは標準モジュールに定義されています。今回は、標準モジュールのSimpleHTTPRequestHandlerクラスを継承して、必要最小限のメソッドだけを定義してみましょう。

新しく定義するハンドラクラスの名前をSimpleAppServerとします。このクラスの中で、リクエストを受け取るためのハンドラメソッドを定義します。GETリクエストを受け取るためにはdo_GET()メソッドを定義します。以下がクラス定義とdo_GET()メソッドの定義です。

List04 do_GET()メソッド(simpleappserver.py)

```
                    ⋮
class
SimpleAppServer(SimpleHTTPServer.SimpleHTTPRequestHandler):

    static_dirs=['/static', ]

    def do_GET(self):
        """GETリクエストを処理する"""
        for sdir in self.static_dirs:
            if self.path.startswith(sdir):
                SimpleHTTPServer.SimpleHTTPRequestHandler.     ❶
                            do_GET(self)
                return
        path=self.path[1:]
        i=path.rfind('?')
        if i>=0:
            path, query=path[:i], path[i+1:]
        else:
            query=''
        self.handle_query(path, query)
                    ⋮
```

static_dirsというアトリビュートには、画像やCSSのような静的なファイルを配信するためのパスをリストに登録してあります。リクエストを受けたときに、URLのパスがこのリストに登録されているものと一致するかどうかを調べて、処理を振り分けるために利用します。

do_GET()というメソッドがGETリクエストを受け取り、処理をするためのメソッドです。メソッドの初めでは、リクエストのパス(self.path)を調べて静的なファイルを配信すべきかどうかを判別しています。static_dirsアトリビュートに登録されている特別なパスから始まるURLについては、SimpleHTTPRequestHandlerのdo_GET()メソッドに処理を渡すことで静的なファイルをレスポンスとして返しています(❶)。

リクエストが特別なパスから始まらない場合は、関数を呼び出します。GETリクエストではURL(パス)にクエリが記載されていることがあります。クエリがあったら抽出し、関数を呼び出すメソッドhandle_query()を呼び出します。

do_POST()メソッド

次に、POSTリクエストを処理するためのメソッドを見てみましょう。POSTリクエストでは、リクエスト本文にクエリが記載されています。そのため、リクエスト本文を読み込んでクエリを解釈する必要があります。

List05 do_POST()メソッド(simpleappserver.py)

```
                    ⋮
    def do_POST(self):
        """POSTリクエストを処理する"""
        length=self.headers.getheader('content-length')
        try:
            nbytes=int(length)
        except (TypeError, ValueError):
            nbytes=0
        data=self.rfile.read(nbytes)
        self.handle_query(self.path, data)
                    ⋮
```

まず、ヘッダからリクエスト本文の長さを取得しています。リクエスト本文は、self.rfileというファイル風のオブジェクトに格納されています。あらかじめ取得したリクエストの長さを使い、read()メソッドでリクエスト本文

を読み込んでいます。クエリを読み込んだ後は、do_GET()と同じようにhandle_query()メソッドを読んで関数呼び出しを行います。

なお、このPOSTリクエストの処理では、静的なファイルに関する処理を考慮していません。

リクエストに合わせて関数を呼び出す

このWebアプリケーションサーバでは、リクエストに合わせて呼び出す関数を登録しておく、という仕様になっていました。ハンドラクラスを定義しているモジュールのトップレベルに、関数を記録するための辞書オブジェクトと登録用の関数を定義してあります。

以下がそのコードです。登録用の関数expose()では、引数として関数オブジェクトそのものを受け取ります。たとえば、foo()という関数を登録したい場合には、関数定義の直後で**expose(foo)**というようなコードを書きます。もちろん、expose()関数はあらかじめインポートしておく必要があります。

expose()関数の内部では、引数で関数名の指定がない場合は、関数オブジェクトの**func_name**アトリビュートを使って関数名を取り出し、関数登録用の辞書を更新しています。

List06 expose()関数（simpleappserver.py）

```
# coding: utf-8

import BaseHTTPServer
import SimpleHTTPServer
import cgi
from httphandler import Response

funcs={}
def expose(func, func_name=''):
    """
    リクエストに反応して呼び出される関数を追加する
    """
    if not func_name:
        func_name=func.func_name
    if func_name=='index':
        func_name=''
    funcs.update({func_name:func})
    return func
                    ⋮
```


do_GET()とdo_POST()では、リクエストに合わせて関数を呼び出すために別のメソッドを呼び出していました。クエリを受け取りメソッドを呼び出す処理は、GET、POSTとも共通しているため、別のメソッド、handle_query()に処理をまとめています。

以下がhandle_query()メソッドの定義です。引数として、パス、およびクエリ文字列を受け取ります。

まず簡単にhandle_query()メソッドの挙動を説明しましょう。expose()関数を使ってfoo()という関数を公開用に登録してあるとすると、~/fooというパスを持つリクエストをfoo()関数に受け渡すという動作をします。

クエリがある場合は、クエリを分解して引数として関数に渡します。クエリはPythonの辞書型と同じようにキーと値のペアで渡されます。これを、キーワード引数（引数名と値のペア）に変換して関数に渡します。また、~/foo/bar/1のように関数名を示す文字列の後にパスか続いていたら、引数名を持たない引数として関数に渡します。これがhandle_query()の大まかな動きです。

List07 handle_query()メソッド（simpleappserver.py）

```
                    ⋮
    def handle_query(self, path, query):
        """
        クエリ付きのGET, POSTリクエストをハンドリングする
        """
        args=[]
        path=path[1:]
        if path.find('/') != -1:                                  ❶
            args=path.split('/')[1:]
            path=path.split('/')[0]
        qdict=cgi.parse_qs(query, keep_blank_values=True)
        for k in qdict.keys():
            if isinstance(qdict[k], list) and len(qdict[k]):
                qdict[k]=unicode(qdict[k][0], 'utf-8', 'ignore')
            else:
                qdict[k]=unicode(qdict[k], 'utf-8', 'ignore')
        if path in funcs.keys():
            qdict.update({'_request':self})
            print args, qdict
            resp=funcs[path](*args, **qdict)                      ❷
```

```
            self.send_response(resp.status,
resp.status_message)
            self.wfile.write(str(resp))
        else:
            self.send_error(404, "No such handler function ( ❸
                    %r)" % path)
                    ⋮
```

メソッドの内部を見ていきましょう。冒頭部分で、まずパスを分割しています（❶）。スラッシュで区切られた最初の部分のみを関数名として取り出し、もし複数の部分があれば関数に渡す引数リストとして保存しておきます。

その後は、cgiモジュールのparse_qs()関数を使ってクエリを解析しています。結果として返ってくる辞書オブジェクトを、関数へ引数として渡すためにローカル変数に保存します。

その後、expose()関数で管理しているfuncsオブジェクトを使い、リクエストに対応する関数を探し出します。関数は辞書の形式で格納されていますので、パスを元にキーがあるかどうかを判別します。キーの検査をしているifブロックの中に、特定のキーに相当する関数を呼び出すコードがあります（❷）。funcs[path](..)という見慣れない書き方がしてありますが、この部分では**辞書に登録されている関数オブジェクトに対して関数呼び出し**を行っています。

関数呼び出しを行っている部分の引数でも、見慣れない書き方をしています。アスタリスクが1つ付いたargsというオブジェクトは、パスをスラッシュで分割したリストです。引数にアスタリスクを1つ付けリストを渡すと、リストの項目が分割されて引数に渡されます。また、アスタリスクが2つ付いたqdictはクエリを抽出した辞書です。辞書にアスタリスクを2つ付けると、辞書のキーと値がキーワード引数になって関数に引き渡されます。このようにして、パスやクエリを関数の引数として渡しています。なお、キーワード引数として渡す辞書には、ハンドラクラスのインスタンスメソッドを渡しています。このインスタンスメソッドには、リクエストのヘッダなどの有用な情報があるためです。

関数からは、Responseクラスのインスタンスオブジェクトが返ってきます。インスタンスオブジェクトを元に、ステータス行とレスポンス本文を送ります。

また、リクエストのパスに該当する関数が見つからなかったときは、ステータスとして404を返しエラーとして扱います（❸）。

サーバの起動

最後に、Webアプリケーションサーバを起動するための関数を定義します。この関数をインポートして呼び出すと、Webアプリケーションサーバが起動し、Webブラウザでアクセスできるようになります。

今回作ったWebアプリケーションサーバを使うには、次のようなステップを踏みます。

①リクエスト経由で呼ばれる関数を登録する

expose()関数を使って、Webアプリケーションサーバに関数を登録します。

②Webアプリケーションサーバを起動する

Webアプリケーションサーバに起動用の関数test()を呼び出します。

起動用関数は、Webアプリケーションの関数などを定義したスクリプトファイルから呼び出すことになるでしょう。つまり、Webアプリケーションの関数と、起動用の関数を呼び出す部分がコードとして記載された「Webアプリケーション起動用スクリプト」ができ上がるわけです。

List08 test()メソッド (simpleappserver.py)

```
                ⋮
def test (HandlerClass = SimpleAppServer,
         ServerClass = BaseHTTPServer.HTTPServer) :
    SimpleHTTPServer.test (HandlerClass, ServerClass)
```

今回作るWebアプリケーションのコードはこれだけです。コードの総行数は70行強しかありません。Pythonの標準モジュールを使うと、初歩的な機能を持ったWebアプリケーションサーバがたった70行で書けてしまうのです。

Webアプリケーションサーバと CGIの違い

これまで作ってきたWebアプリケーションはCGIの仕組みを使って動いていました。Pythonのスクリプトファイルを設置し、スクリプトファイルを指すURLをリクエストとして送ると、Webサーバがスクリプトをプログラムとして起動します。この場合、スクリプトファイルはリクエストが送られるごとに、毎回起動することになります。スクリプトファイル上に定義したローカル変数などは毎回初期化されることになります。

Webアプリケーションサーバは CGIとは違った仕組みで動きます。Webサーバが起動するときに、リクエスト経由で呼び出す関数を登録します。関数は1つのファイルにまとめて書いてあっても構いません。また、Webサーバが起動している間は、関数などのオブジェクトがメモリ上に存在し続けます。

このように、CGIとWebアプリケーションサーバの間には、Webアプリケーションを動かす仕組みに大きな違いがあります。この違いを利用して、ちょっと面白い実験をしてみましょう。データベースやファイルのような仕組みを使わずに、アクセス回数を数えるカウンタを作ってみましょう。

以下がそのコードです。先ほど作ったWebアプリケーションサーバのハンドラクラス、Requestクラス、テンプレートエンジンSimpleTemplateを活用しています。これをcgi-binフォルダの中に保存します。

List09 countertest.py

```
#!/usr/bin/env python
# coding: utf-8

from simpleappserver import expose, test
from httphandler import Response
from simpletemplate import SimpleTemplate

@expose                                                        ❶
def index(_request, d={'counter':0}):
    body="""<html><body><p>${counter}</p></body></html>"""
    res=Response()
    t=SimpleTemplate(body)
    body=t.render(d)
    d['counter']+=1
    res.set_body(body)
    return res

if __name__=='__main__':
    test()
```

index()という関数は、スラッシュで終わる**インデックスアクセス**を受け付けるための関数です。この関数の直前の行に、アットマーク（@）で始まる見慣れない表記があります（❶）。これは**関数デコレータ**と呼ばれています。この表記は、「expose()という関数に直後の関数オブジェクト（index）を渡して呼び出す」という内容の処理をします。ちょうど、「index=expose(index)」というコードと同じ意味になります。

expose()というのは、Webアプリケーションサーバのソースコードに定義された呼び出し関数を登録するための関数です（p.242）。この関数には関数オブジェクトを渡す約束になっていました。つまり、expose()関数を関数デコレータとして使うと、Webアプリケーション側に関数を登録するという処理が行えるわけです。

関数内の処理はとても簡単です。辞書の**counter**というキーを表示するだけのテンプレートを定義し、レスポンスとして返しています。テンプレートエンジンに渡す辞書は、引数dに代入された辞書です。関数が呼び出されるたび、この辞書のcounterというキーに対応する値を1ずつ加算していきます。

Pythonでは、引数に定義された辞書やリストのような書き換え可能オブジェクトは、読み込み時に一度だけ初期化されます。今回作ったWebアプリケーションサーバでは、一度登録した関数などのオブジェクトはサーバが動いている間はずっとメモリ上に残り続けます。つまり、関数が呼び出されるごとに辞書のキーに相当する値が加算されていくわけです。テンプレートではキーの値を表示するようになっていますので、ブラウザをリロードするたびに値が加算されていくはずです。

countertest.pyの起動

CGIHTTPServerが起動している場合は終了させて、countertest.pyを起動してみてください。このファイルの最後にはWebアプリケーションサーバを起動するための関数が書き込まれていますので、スクリプトを走らせると

Webアプリケーションサーバが走り出します。スクリプトを起動したら、Webブラウザで「http://127.0.0.1:8000/」というURLにアクセスします。ブラウザをリロードするたびに、数値が1つずつ増えていくはずです。Webアプリケーションサーバを終了し、再度起動するとカウンタの数値が0に戻ります。再起動することによって辞書の内容が初期化されるためです。

図01 カウンターの様子

CGIのように、毎回スクリプトが立ち上がるような仕組みを使う場合は、カウンタのような機能を実現するには、カウンタの値をファイルやデータベースに保存しておく必要があります。Webアプリケーションサーバでは、関数などのオブジェクトがメモリ上に長く残っているため、このようなトリックが利用できるのです。

CHAPTER 14

Webアプリケーションと認証

Webアプリケーションの中には、ユーザ名とパスワードを使って**ログイン**を行ってから機能が利用できるようになるものがあります。認証を行うことによって、誰がWebアプリケーションを使っているかが明確になり、権利を持った利用者にのみ、特定の機能を利用させる、といったことが可能になります。ここでは、Webアプリケーションで認証の処理を行うための方法について考えてみたいと思います。

14-01 認証の基本

私たちは日常生活でさまざまな**認証**を行います。たとえば、キャッシュカードを使いキャッシュディスペンサーで現金を下ろすときには、暗証番号を使った認証を行います。家のドアを開け閉めするとき、鍵を使うのも一種の認証と言えるかもしれません。このように、自分以外の人間やモノに対して、何らかの手段を使って自分自身であることを認めてもらう行為のことを認証と呼びます。

認証を行うときには、他人や物体、プログラムなどに自分自身であることを認めてもらう必要があります。そのためには、第三者にも客観的に認めてもらえる「モノ」を使う必要があります。暗証番号、パスワード、鍵などはそのような「モノ」の例です。

Webアプリケーションでは、たいてい**ユーザ名**と**パスワード**を使って認証を行います。たとえば、ブログのようなWebアプリケーションには、2種類のユーザがいます。ブログを見る人と書く人です。すべてのユーザがブログの記事を書き換えられるのでは問題があります。どのユーザがブログを読む人

で、どのユーザが書く人であるかをブログのWebアプリケーションが見分ける必要があります。そのために、ブログを**書く人はユーザ名とパスワードを使ってブログにログイン**してから、記事を書くわけです。

この場合は、ユーザ名とパスワードが第三者（Webアプリケーション）にも客観的に認めてもらえる「モノ」として機能していることになります。本人しか知り得ない情報を使って、本人であることを確認するのが認証の基本的な考え方です。

Webアプリケーションでの認証

Webアプリケーションは**クライアントサーバモデル**と呼ばれる仕組みで動きます。クライアントとなるWebブラウザなどと、実際に処理を行うプログラムは離れた位置にあって、ネットワークを通じて通信を行います。

認証に必要なユーザ名やパスワードなどのデータを送るのはクライアントです。それに対して、認証に必要なデータを評価し、正しいデータであることを評価するのはサーバ側にあるプログラムです。Webアプリケーションでは、リクエストとレスポンスが対応を作りながら処理を続けていきます。つまり、認証を行うときにも、リクエストとレスポンスを組み合わせながらデータや結果のやりとりを行う必要があるわけです。実際、Webアプリケーションの認証処理は、フォームを使った入力の遷移ととても似た形で進んでいきます。

認証状態の継続

もう1つ、Webアプリケーションの認証で考えなければならないことがあります。それは**認証状態の継続**についてです。

Webアプリケーションでは、リクエストとレスポンスのペアをやりとりしながら処理を進めていきます。認証を行うため、ユーザ名とパスワードをPOSTメソッドを使ってリクエストとして送ったとしましょう。そのリクエストに対応するレスポンスを作る際には、Webアプリケーションのプログラムは認証に必要なデータを得ることができます。しかし、ログインをしたあと、データを編集するために送ったPOSTのリクエストに認証に必要な情報が含まれていなかったら、プログラム側でどのようにしてログイン済みであることを判断すればよいでしょうか。何らかの方法を使って、リクエストを使ってユーザが認証状態であることを示せないと、毎回ユーザ名とパスワードをクエリを使って送信しなければならない、ということになってしまいます。

実際は、多くのWebアプリケーションでは、リクエストのヘッダの中に何らかの情報を埋め込んで認証状態を継続しています。Webアプリケーションでは、認証情報の送り方と、認証状態の継続の仕方によって、認証を行うための方法が何種類かに分かれます。

14-02　BASIC認証

BASIC認証はWebアプリケーションで最も手軽に利用できる認証方式の1つです。Webアプリケーションなどを使っているとき、ユーザ名とパスワードを入力するダイアログが現れることがあります。このようなダイアログが現れたときが、サーバ側でBASIC認証を要求している合図です。Webブラウザには、BASIC認証に対応するための機能が組み込まれています。サーバの要求に応じて、Webブラウザがダイアログを自動的に表示しているのです。

図01　BASIC認証を要求するときには、Webブラウザが認証用のダイアログを開く

BASIC認証の仕組み

BASIC認証は次のような仕組みで機能します。認証に必要な情報は、クエリではなくヘッダを使ってやりとりされます。また、認証の必要なパスにアクセスするために、少なくとも2回のリクエストを送っているということに

なります。

1-A）クライアントが認証の必要なパスにリクエストを送る
Webサーバからのリクエストが認証の出発点になります。

1-B）サーバが401というステータスのレスポンスを返す
要求されたリクエストに応答するためには認証が必要であることを、Webブラウザに知らせるために特別なステータス番号を送信します。

2-A）クライアントがダイアログを表示する
その後、ヘッダに認証情報を暗号化して埋め込み、サーバに送信します。実際には、以下のようなヘッダがWebブラウザからサーバに送られます。

```
Authorization: Basic（暗号化したユーザ名とパスワード）
```

2-B）サーバ側で認証情報を解釈し、結果を再送信する
ヘッダの認証情報を復号化（暗号の解除）し、正しい認証情報かどうかを判別します。正しい認証情報であることが分かれば、正しいページを返します。もし正しくない場合は、再度1-Bに戻りステータス401のレスポンスを返します。

BASIC認証の継続

2-AでWebブラウザが送るヘッダには**Authorization**というヘッダ名が付いています。一度認証を行うと、Webブラウザは同じホストのリクエストのヘッダに2-Aで送ったものと同じヘッダを送り続けます。このようにして、ユーザ名とパスワードを一度入力しただけで、認証状態を継続できるわけです。

このヘッダは、Webブラウザを終了するまで送り続けられます。このような挙動は一見便利なようですが、別の問題を引き起こすこともあります。Webブラウザが勝手に認証情報を送り続けてしまうため、いわゆる**ログアウト**に相当する処理ができないのです。

ログアウトに相当する処理を実行するためには、Webブラウザ側でステータス401のコードを再度送信し、Webブラウザが再度ダイアログを表示したときにキャンセルボタンを押すなどする必要があります。つまり、ユーザ側で明示的にWebブラウザに入力した認証情報を消去する必要があるわけです。

認証情報の暗号化方式

Webブラウザが送信するリクエストのヘッダ部分には、暗号化したユーザ名とパスワードが記載されます。Webアプリケーションのプログラム側では、この情報を復号化してユーザが入力したユーザ名とパスワードを取り出し、正しい認証情報かどうかを確かめます。

BASIC認証では、「BASE64」と呼ばれる暗号化の仕組みが使われます。BASE64は、入力された文字列などのデータを数値とアルファベットの組み合わせに変換します。この変換方式のことを**BASE64エンコード**と呼びます。BASE64エンコードされた文字列は、一定のルールを使うことで元のデータを取り出すことができます。BASE64はとても手軽な暗号化方式です。BASIC認証だけでなく、電子メールの添付ファイルの暗号化などにも利用されています。

Pythonの標準モジュールには**base64**というモジュールがあり、このモジュールを使うとBASE64エンコードの暗号化と復号化が行えます。以下の例では、インタラクティブシェルを使ってbase64の暗号化と復号化を試してみています。

●base64モジュールの利用例：

```
>>> import base64
>>> e=base64.encodestring('abcdefg')
>>> print e
YWJjZGVmZw==

>>> print base64.decodestring(e)
abcdefg
```

14-03 ダイジェスト認証

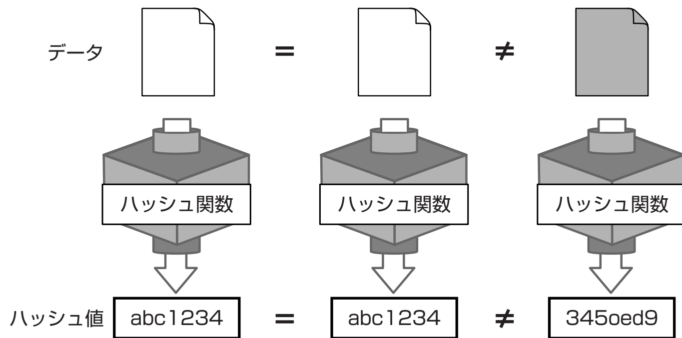
BASIC認証はとても手軽な認証方式ですが、大きな欠点があります。ヘッダに埋め込むユーザ名とパスワードを容易に抜き出せるのです。認証情報が可逆的な暗号化方式で暗号化されているため、Webブラウザからサーバに送信されるヘッダをのぞき見ることができれば、たやすくユーザ名とパスワードを抜き出すことができるのです。

この欠点を解消するため、**ダイジェスト認証**と呼ばれる認証方式が考え出されました。ダイジェスト認証では、ユーザ名やパスワード自体がやりとりされることがありません。ユーザ名やパスワードの代わりに、**メッセージダイジェスト**と呼ばれる一種の「合言葉」を使います。そのため、より安全に認証を行えるわけです。

メッセージダイジェストとハッシュ

メッセージダイジェストには**ハッシュ**を使って認証を行います。ハッシュとは、文字列などのデータから作る値のことを指します。ハッシュを作るための方式は何種類かあって、この方式のことを**ハッシュ関数**と呼びます。また、ハッシュ関数を使って得た値を**ハッシュ値**と呼びます。ハッシュは、Webはもちろんネットワークの世界でとても重要な概念です。

図02 同じデータから作ったハッシュ値は必ず同じになる



"ABC"という文字列データを、「A」というハッシュ関数を使ってハッシュ化するとします。すると、いつも同じハッシュ値を得ることができます。逆に、違うデータを与えると、ハッシュ関数はまったく違ったハッシュ値を作ります。

また、ハッシュ値と元の文字列の間にはほとんど関連性がなく、ハッシュ値から元の値を復元することはまず不可能です。

ハッシュ関数にはいくつかの種類があります。よく利用されるのが**MD5**や**SHA-1**と呼ばれるハッシュ関数です。Pythonの標準モジュールを使うと、MD5とSHA-1といったハッシュ関数を利用したハッシュを生成できます。

インタラクティブシェルを使ってmd5モジュールを使ってみましょう。new()関数を使い、md5オブジェクトを生成します。その後、hexdigest()メソッドを使い、ハッシュ値を16進数文字列で表示しています。

● MD5によるハッシュの作成例：

```
>>> import md5
>>> m=md5.new('abcde')
>>> print m.hexdigest()
ab56b4d92b40713acc5af89985d4b786
```

パスワードのような比較的短い文字列だけでなく、ファイルやテストといった長いデータを使ってもハッシュ値を作ることができます。データが少しでも異なるとまったく違ったハッシュ値が出力されます。

このような性質を使うと、ハッシュを認証に利用できるのです。つまり、ハッシュ値を比べるだけで、元のデータが同じかどうかを調べることができるわけです。

ダイジェスト認証の仕組み

ダイジェスト認証が行われる仕組みはBASIC認証とよく似ています。ユーザ名やパスワードそのものに近いデータがやりとりされないこと、セキュリティを保つためにヘッダ上で付加的なデータがやりとりされること、などがBASIC認証と異なります。実際にやりとりされるのはハッシュ値なので、サーバ側にユーザ名やパスワードを保存する必要がないのもダイジェスト認証の特徴の1つです。

ダイジェスト認証は以下のような仕組みで機能します。

1-A) クライアントが認証の必要なパスにリクエストを送る

Webサーバからのリクエストが認証の出発点になります。BASIC認証と同じです。

1-B) サーバが401というステータスのレスポンスを返す

要求されたリクエストに応答するためには認証が必要であることを、Webブラウザに知らせるために特別なステータス番号を送信します。BASIC認証との違いは、ヘッダを使って付加的なデータを送信する点です。

2-A) クライアントがダイアログを表示する

その後、ヘッダにユーザ名やパスワードから作ったハッシュ値を埋め込み、サーバに送信します。実際には、以下のようなヘッダがWebブラウザからサーバに送られます。BASIC認証で**Basic**となっていた部分が**Digest**となっています。

```
Authorization: Digest（ユーザ名やハッシュ値などが続く）
```

2-B) サーバ側で認証情報を確認し、結果を再送信する

ヘッダのユーザ名、ハッシュなどを評価し、正しい認証情報かどうかを判別します。

2-Aで送信するレスポンスには、ヘッダに以下のような追加の情報が記載されています。**nonce**とはWebサーバが作るランダムな文字列です。Webサーバは、この文字列を次回のリクエスト時に送信しなければなりません。**algorithm**でハッシュ関数を指定しています。

```
WWW-Authenticate: Digest realm="Secret Zone",
   nonce="Ny8yLzIwMDIgMzoyNjoyNCBQTQ",
   algorithm=MD5, qop="auth"
```


ダイジェスト認証の継続

Webブラウザを使って一度ダイジェスト認証を行うと、WebブラウザはWebサーバにアクセスするたび、リクエストに認証用のヘッダを追加します。Webサーバ側では、認証用のヘッダを確認できるので、認証状態が継続できます。認証用のヘッダはWebブラウザを終了するまで送り続けられます。ですので、いわゆるログアウトをするためにはWebブラウザを終了するか、認証ダイアログを再度表示してユーザ名と空のパスワードを入力するなどする必要があります。

14-04　フォームを使った認証

BASIC認証やダイジェスト認証はとても手軽に利用できる認証方式です。多くのWebサーバが対応していますし、Webブラウザが自動的にヘッダに認証情報を送信してくれるので、認証状態を継続できます。

半面、認証状態が勝手に継続してしまうため、いわゆるログアウトが面倒なのが難点です。また、ブラウザを終了すると認証情報が消えてしまいます。そのため、Webブラウザを起動するたびに再度ログインする必要があるのです。

ログインとログアウトを明示的に行ったり、ブラウザを終了しても一定期間ログイン状態を保持できれば便利な場合があります。ソーシャルネットワークのような一般的なWebアプリケーションではそのような機能を実現しているものもあります。

このように、ログアウトやログインの期間を変えるなどといった認証の要求に応えるために**フォーム認証**が利用されることがあります。BASIC認証のようにブラウザが表示するダイアログに認証情報を入力するのではなく、Web上のフォームを使ってユーザ名やパスワードといった認証情報を入力するのです。

フォーム認証の遷移

フォーム認証では、認証情報の入力にフォームを使います。フォームに入

力されたユーザ名やパスワードを受け取るのはWebアプリケーションのプログラムです。つまり、フォームを使ったWebアプリケーションと同様に処理が進んでいくのです。入力されたユーザ名やパスワードに入力ミスがあった場合などには、誤りであることをユーザに知らせなければなりません。ログインを行うときの遷移について考える必要があります。

フォーム認証時の**遷移**については、一般的な解答があるわけではありません。そこでここでは、できるだけ「気の利いた」遷移について考えてみることにしましょう。

まず、ユーザが認証情報を間違えて入力した場合のことを考える必要があります。ユーザ名やパスワードを間違えて入力した場合は、間違いであることを表示しつつフォームの再表示をすると便利なはずです。ただしこの場合、ユーザ名のみ残してパスワードは消去します。また、ユーザ名とパスワードどちらか一方があっていても、ただ単に「間違っている」と表示します。どちらか一方が合っていることをユーザに分かるように表示してしまうと、パスワード破りに利用される可能性があるのです。

フォーム認証の場合は、ログインフォームを表示する専用のURL上の**パス**を設置して利用します。無認証状態でユーザがWebアプリケーションにアクセスしたときには、このパスにリダイレクトすることでフォームを表示します。ログイン後は、最初に表示しようとしたURLに再度リダイレクトするようにしましょう。ユーザはもともと最初にアクセスしたページを見たかったわけですから、自動的にそのURLに移動すれば利便性が増すはずです。

ログイン後、リダイレクトをするには、最初にいたページのURLの情報を保存しておく必要があります。GETリクエストのクエリとして渡すか、フォームにhiddenフィールドとして埋め込んでおくとよいでしょう。

フォーム認証での認証状態の継続

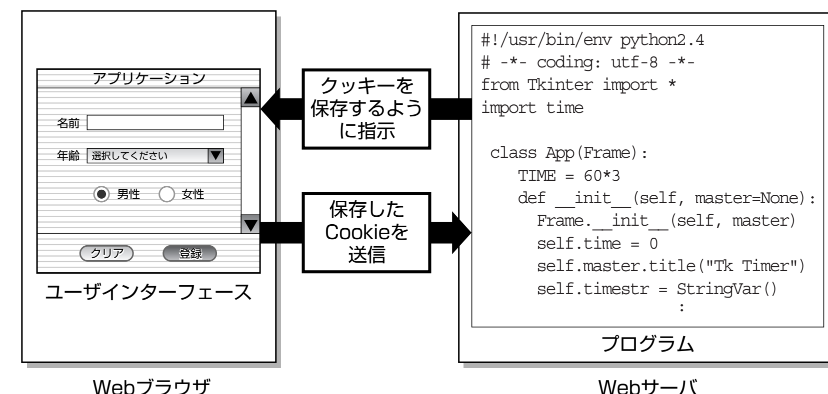
Webアプリはリクエストとレスポンスを繰り返すことで処理を進めていきます。このため、「ログイン中である」といった**状態を維持**するには、リクエストを使って付加的な情報を送り続ける必要がある、ということはすでに説明したとおりです。BASIC認証やダイジェスト認証では、Webブラウザがリクエストのヘッダに自動的に認証情報を付加します。フォーム認証の場合は、認証を継続するために必要な情報を受け渡す方法も自前で用意する必要があります。

フォームを使った認証では、**Cookie**（**クッキー**）を使って認証状態を維持する手法がよく採用されます。Cookieとは、Webブラウザ上にさまざまな値を保存するために利用される仕組みです。簡易なデータベースのようなもので、Webサーバのドメインやパスなどと関連付けて、Webブラウザ上にキーと値を保存できます。

Cookieに値を保存するためには、WebサーバやWebアプリケーションの送り出すレスポンスに必要な情報を書き出します。Cookieを保存、更新、削除するためにWebサーバが送り出すヘッダがあらかじめ決められています。Webブラウザはレスポンスのヘッダを見て、必要があればCookieのデータベースを更新します。

Webブラウザは、必要があればCookieの情報をWebサーバやWebアプリケーションに伝えます。これにはリクエストヘッダが使用されます。

図03 Cookieの仕組み



つまり、Cookieを使うと、Webアプリケーションからの指令で、任意のデータをWebブラウザ上に保存できるわけです。保存したデータはヘッダを通じてWebアプリケーションで利用できます。大量のデータを保存することはできませんが、ちょっとした文字列であればCookieを使って保存できます。この方法を使うと、BASIC認証やダイジェスト認証で使われていたのと似た方法を使って、ログイン状態を継続できます。

Cookieを使って認証状態を継続する手法の利点は、認証状態をコントロールできる、ということです。Cookieはレスポンスのヘッダを使うことで簡単にコントロールできます。ログイン中はCookieの特定の文字列に「合言葉」となる文字列を保存しておき、ログアウトしたいときには合言葉を消せばよいわけです。また、Webブラウザを終了してもCookieを保存されたままにできます。Webブラウザを終了してもログイン状態を維持できるわけです。

PythonとCookie

Pythonの標準モジュールには**Cookie**というモジュールがあります。このモジュールを使うと、クライアントとWebサーバ間でやりとりされるヘッダを手軽に扱えます。Cookieモジュールにはいくつかのクラスが定義されています。**SimpleCookie**というクラスを使って、クライアントとサーバの間でやりとりされている「生のCookie」を覗いてみましょう。

まずはSimpleCookieを使って、Webサーバからクライアントに送られるCookieのヘッダを見てみましょう。SimpleCookieは、Pythonの辞書オブジェクトのように扱えます。インスタンスを作った後、辞書のようにキーを使って値を登録します。キーを入れ子にして、**path**（Cookieが有効になるパス）、**expires**（Cookieの有効期限）などを指定することもできます。

インタラクティブシェルを使ってSimpleCookieを使ってみましょう。以下の例では、まず**foo**というキーに文字列を設定しています。また、expiresを指定してCookieの有効期限を遠い未来に指定し、長い期間残るCookieを設定しています。いったんCookie用のヘッダを表示した後、さらに**bar**というキーを設定して、再度ヘッダを表示しています。

● コマンドライン Cookieモジュールの利用例①：

```
>>> from Cookie import SimpleCookie
>>> c=SimpleCookie()
>>> c['foo']='cookie value'
>>> c['foo']['expires']='Thu, 1-Jan-2030 00:00:00 GMT'
>>> print c.output()
Set-Cookie: foo="cookie value"; expires=Thu, 1-Jan-2030 00:00:00
GMT
>>> c['bar']='another cookie value'
>>> print c.output()
Set-Cookie: bar="another cookie value"
Set-Cookie: foo="cookie value"; expires=Thu, 1-Jan-2030 00:00:00
GMT
```

ブラウザが**Set-Cookieヘッダ**を受け取ると、有効期限やパスなどを考慮してCookieの値を保存します。ブラウザが保存したCookieの値は、必要に応じてリクエストの**Cookieヘッダ**に記載されます。Webアプリケーションのプログラムでは、このヘッダを解釈して、Cookieの値を得ることができます。このようにして、Cookieの値をやりとりするのです。

SimpleCookieクラスを使うと、リクエストのヘッダを解釈してPythonのオブジェクトに変換できます。変換したオブジェクトは辞書のように扱えます。キーを指定してCookieの値を読み込むことができるのです。

インタラクティブシェルで試してみましょう。クライアントから送られてくるCookieヘッダを使ってSimpleCookieインスタンスを初期化します。すると、Cookieの値が辞書風のオブジェクトになって返ってきます。キーの値として格納されているのは**Morselオブジェクト**です。値を取るためには**value**というアトリビュートを参照します。他にexpiresやpathといったアトリビュートを使うこともできます。

● コマンドライン Cookieモジュールの利用例②：

```
>>> from Cookie import SimpleCookie
>>> c2=SimpleCookie('Cookie: foo=string1; bar=string2;')
>>> c2.keys()
['foo', 'bar']
>>> print c2['foo'].value
string1
```

14-05　フォーム認証の機能を作る

実際にフォーム認証の仕組みを作ってみましょう。これまで作ってきたテンプレートエンジンやWebアプリケーションを活用して、フォーム認証の仕組みを作ります。

先ほど作ったWebアプリケーションでは、モジュールに定義した関数に、

デコレータを設置するとリクエストに応答するように関数を加工できました。同じように、デコレータを使って**アクセス時に認証が必要な関数**が設定できると便利です。今回はデコレータを活用して、アクセスするためにフォーム認証が必要な関数を作ってみましょう。

今回は必要なすべてのコードを1つの「authentication.py」というモジュールファイルに収めます。

認証フォーム一式を作る

まずはフォーム認証に必要な一連の遷移を作る必要があります。ウィジェットとバリデータを活用して、間違いがあったら再入力を促すような使いやすい認証フォームを作りましょう。

まずは必要なライブラリのインポートとフォームの定義をします。フォームの定義には、先ほど作成したウィジェットとバリデータを使います。ユーザ名とパスワードは、アルファベットと数値のみ受け付けるようにします。

List01 フォームの定義（authentication.py）

```
#!/usr/bin/env python
# coding: utf-8

from simpleappserver import expose, test
from httphandler import Response
from simpletemplate import SimpleTemplate

from validators import NotEmpty, RegexValidator
from widgets import Text, Submit, Form

editforms=(Text('username', u'ユーザ名',
           validators=(NotEmpty(),
           RegexValidator(r'[A-Za-z¥d]')),),
           Text('password', u'パスワード',
           validators=(NotEmpty(), RegexValidator(
                  r'[A-Za-z¥d]')),),
           Submit('submit', u'ログイン'))
loginform=Form(editforms, {'action':'/login', 'method':'POST'})

base_body="""<html><body>%s</body></html>"""
                              ⋮
```

次にフォームを表示するための関数を作ります。Webアプリケーションに定義されたexposeデコレータを使い、Webブラウザでアクセスできるようにします。~**/login_form**というURLにアクセスすると、フォームが表示されます。

List02 login_form()関数（authentication.py）

```
                              ⋮
@expose
def login_form(_request, values={}, errors={}):
    body=base_body % ('${form.display(values, errors)}')
    res=Response()
    t=SimpleTemplate(body)
    values['password']=''
    body=t.render({'form':loginform, 'values':values,
                   'errors':errors})
    res.set_body(body)
    return res
                              ⋮
```

この関数は、フォームの入力に間違いがあった場合などに再度呼ばれます。そのときのために、前回のリクエストで入力された値とエラーを辞書で受け取っています。辞書はテンプレートエンジンに渡して、必要があればフォームに値やエラーを埋め込んで表示します。

次に、フォームからPOSTされたリクエストを受け取る関数を作ります。この関数は、ウィジェットに定義済みです。

List03 login()関数（authentication.py）

```
                              ⋮
from Cookie import SimpleCookie
import md5

fixeduser='user'                                        ❹
fixedpass='pass'

@expose
def login(_request, username='', password=''):
    res=Response()
    values, errors=loginform.validate({'username':username,
```

```
                                       'password':password})
    if errors or fixeduser!=username or fixedpass!=password: ❶
        return login_form(_request, values, errors)

    c=SimpleCookie()
    m=md5.md5(username+':'+password)                          ❷
    c['authhash']=m.hexdigest()                               ❸
    c['authhash']['expires']='Thu, 1-Jan-2030 00:00:00 GMT'
    res.set_header(*c.output().split(': '))
    res.status=302
    res.set_header('Location', '/')
    res.set_body('')
    return res
                              ⋮
```

この関数では2種類の処理をしています。1つ目は、フォームからのPOSTリクエストを受け付け、フォームに入力された値が空であったり、英数字以外の文字列が含まれていないかどうかを調べるバリデーションチェックです。2つ目は、入力されたユーザ名とパスワードが正しいかどうかを調べる処理です。正しいユーザ名（fixeduser）とパスワード（fixedpass）はモジュールのアトリビュートに固定で設定しておきます（❹）。本来なら、データベースに登録したユーザ名とパスワードを問い合わせるような実装になるはずです。

入力に間違いがあった場合は、login_form()関数を呼び出してフォームを再表示します（❶）。正しいユーザ名とパスワードが入力された場合は、SimpleCookieを使ってCookieに認証用の値を設定するようヘッダを記載し、レスポンスを返します。クッキーの値は、ユーザ名とパスワードをコロン（:）で連結した文字列から生成したMD5のハッシュ文字列です（❷）。Cookieに値を設定するとともに、リダイレクトを行っています（❸）。

念のため、ログアウト用の関数も作っておきましょう。Cookieに入っている値を空にして、メッセージを表示するだけの単純な関数です。

List04 logout()関数（authentication.py）

```
                              ⋮
@expose
def logout(_request):
    body=base_body % ('<p>Logged out</p>')
    res=Response()
    c=SimpleCookie()
    c['authhash']=''
    res.set_header(*c.output().split(': '))
    res.set_body(body)
    return res
                              ⋮
```

ログインチェック用の仕組みを作る

次に、ログイン状態を確認するためのデコレータを準備しましょう。以前に作ったexpose()は関数でしたが、今回はクラスを作ります。今回のデコレータでは、汎用性を高めるためログイン状態を確認する関数と、ログイン用のパスを引数で指定できるようにしたいからです。デコレータ指定時に引数を渡して、デコレータの挙動をコントロールできるようにしておけば、チェック用の関数を入れ替えたり、フォームを表示するURLを変更したりできます。デコレータで引数を受け取れるようにするためには、クラスを作る必要があるのです。

List05 secured_exposeクラス（authentication.py）

```
                              ⋮
class secured_expose(object):
    """
    認証付きのリクエストハンドラ関数を定義するためのデコレータクラス
    """

    def __init__(self, checkfunc, loginpath='/login_form'):
        self.loginpath=loginpath
        self.checkfunc=checkfunc

    def __call__(self, func):
        def wrapper(_request, *args, **kws):
            if self.checkfunc(_request):
                return func(_request=_request, *args, **kws)
            else:
                res=Response()
                res.status=302
```

```
                res.set_header('Location', self.loginpath)
                res.set_body('')
                return res
        expose(wrapper, func_name=func.func_name)
        return wrapper
                            ⋮
```

まず、デコレータ指定時に受け取りたい引数をクラスの初期化メソッド__init__()に指定します。その後、実際にデコレータとして機能するメソッド__call__()を定義します。__call__()は、インスタンスオブジェクトに直接丸カッコを記述したときに呼ばれる特殊メソッドです。

なお、この例では__call__()メソッドの内部にwrapper()という入れ子の関数を定義しています。入れ子の関数内部で、デコレータ指定された関数を場合分けして実行するのが狙いです。

wrapper()関数内部では、まずリクエストの状態を受け取り認証状態をチェックしています。インスタンスの初期化時に渡された関数オブジェクトを呼び出す形でログイン状態のチェックを行っています。

ログイン状態であることが確認できたときには、func()を呼び出してリクエストを処理します。

ログイン状態であることが確認できなかったときには、302のステータス番号を発行してリダイレクトを行います。デコレータに引数として指定されたパスを対象にリダイレクトを行います。

__call__()メソッドの最後では、expose()関数を呼んでいます。関数の第2引数として渡ってくる関数オブジェクトを、URL呼び出しできるようにWebアプリケーションに登録することが目的です。最後に、wrapper()という関数オブジェクト自体を戻り値として返しています。

次に、ログイン状態をチェックする関数を作りましょう。リクエストのCookieを解釈して、ログイン状態を正しく示す値が登録されているかどうかを調べて結果を返します。

List06 checklogin()関数 (authentication.py)

```
                            ⋮
def checklogin(request):
    c=SimpleCookie(request.headers.getheader('Cookie', ''))
    m=md5.md5(fixeduser+':'+fixedpass)
    digest=m.hexdigest()
    if c.has_key('authhash') and c['authhash'].value==digest:
        return True
    else:
        return False
```

関数の内部では、Webブラウザが送信した、Cookieの記載されているヘッダを調べています。ヘッダの内容は、ユーザ名とパスワードから生成したMD5のハッシュです。ユーザ名とパスワードは固定なので、同じハッシュ文字列を作ってCookieの内容と照合しています。もしCookieとプログラム内部で生成した2つのハッシュが同じときには、ログイン状態と見なしてTrue（真）を返します。そうでない場合にはFalse（偽）を返します。

最後に、このデコレータクラスを使って閲覧に認証が必要な関数を作ってみましょう。secured_exposeクラスをデコレータとして指定するだけです。ただし、このときにログインチェック用の関数オブジェクト（checklogin）を引数として渡す必要があります。

このように、デコレータを作るだけで、閲覧に認証が必要な関数ができるのです。

List07 index()関数 (authentication.py)

```
                            ⋮
@secured_expose(checkfunc=checklogin)
def index(_request, foo='', d={'counter':0}):
    body=base_body % ('<p>Logged in!</p>')
    res=Response()
    t=SimpleTemplate(body)
    body=t.render(d)
    d['counter']+=1
    res.set_body(body)
    return res

if __name__=='__main__':
    test()
```

また、このモジュールの最後にWebアプリケーションサーバを起動するためのコードが記述されています。

コードの記述が終了したら、cgi-binフォルダの中に保存してください。

フォーム認証を試す

プログラムができ上がったので「authentication.py」を起動して試してみましょう。

authentication.pyが起動したら、**http://127.0.0.1:8000/**にアクセスします。index()メソッドにアクセスが行くのですが、このメソッドはデコレータによって認証が必要になるように設定されています。無認証状態でこの関数にアクセスすると、デコレータが機能して**~/login_form**にリダイレクトします。

図04 ログインフォームの表示

その後、正しいユーザ名（user）とパスワード（pass）を入力すると、認証用のCookieが設定されてログイン状態になり、index()メソッドにアクセスできるようになります。

図05 ログインに成功した場合

Cookieを設定するとき、有効期限を遠い未来に設定していました。そのため、WebアプリケーションサーバやWebブラウザを終了しても、ログイン状態が継続します。

ログイン状態を解除するには、Webブラウザを使って**~/logout**というURLにアクセスします。するとCookieがクリアされ、ログイン状態でなくなります。再度index()関数にアクセスするためにドキュメントルートのURLにアクセスすると、ログインフォームにリダイレクトされます。

CHAPTER 15

Webアプリケーションとセキュリティ

Webアプリケーションが高機能になると、使いやすさが増します。しかし、Webアプリケーションが高度になって行く一方で、危険性が増すのも事実です。ここでは、Webアプリケーションを作る上で気をつけるべき、セキュリティ上の問題について考えます。

15-01　Webアプリケーションのセキュリティホール

テンプレートエンジンやWebアプリケーションサーバを使うと、高度な機能を持つWebアプリケーションを比較的手軽に作れるようになります。高度な機能を持つWebアプリケーションは便利で使いやすいものです。しかし、機能が高度になっていくにつれ、いろいろな危険性が紛れ込む余地が増えるのも事実です。

開発者が、自分の作ったプログラムに危険性を意図的に埋め込むことはまず考えられないでしょう。そういう意味では、ソフトウエアに潜む危険性は開発者が意図しない形で現れることが多いのかも知れません。このように、開発者が意図しない操作が許されてしまうような欠陥のことを**セキュリティホール**と呼びます。また、本来見えるべきでない情報が第三者に見えてしまうような不具合も同様にセキュリティホールと呼ばれます。

プログラムの欠陥は、動作原理に大きな関わりを持っています。Webアプリケーションに見られるセキュリティホールは、Webアプリケーションの動作原理に大きく関わっています。Webアプリケーションでは文字列操作を頻繁に行います。そのため、セキュリティホールの多くは文字列に関わった形で現れます。

セキュリティホールの例

では、実際にWebアプリケーションに見られる典型的なセキュリティホールを見てみましょう。

まず、簡単な問い合わせフォームを作ってみます。「xsstest.py」というファイル名でcgi-binフォルダに保存します。本書で作ったWebアプリケーションサーバ、テンプレートエンジンなどを使いますので、モジュールファイルを同じ階層に置く必要があります。なお、セキュリティホールの実験のために作るWebアプリケーションですので、お問い合わせフォームとしては機能しません。

List01　xsstest.py

```
#!/usr/bin/env python
# coding: utf-8

from simpleappserver import expose, test
from httphandler import Response
from simpletemplate import SimpleTemplate

htmlbody=u"""<html><body>
<h2>お問い合わせフォーム</h2>
<form>
名前 :<br/>
<input type="text" name="name" value="${name}"/> <br/>
本文 :<br/>
<textarea name="body" cols="40" rows="10">${body}</textarea> <br/>
<input type="submit" name="submit" value="送信" />
</form>
</body></html>"""

@expose
def index(_request, name='', body=''):
    res=Response()
    t=SimpleTemplate(htmlbody)
    body=t.render({'name':name, 'body':body})
    res.set_body(body)
    return res
```

```
if __name__=='__main__':
    test()
```

スクリプトファイルを直接起動するとWebアプリケーションが動き出します。Webブラウザで**http://127.0.0.1:8000/**にアクセスすると、フォームが表示されるはずです。

このWebアプリケーションでは、クエリで名前（name）や本文（body）を受け取ってフォームの中に埋め込むようになっています。本書で作ったWebアプリケーションでは、クエリが関数の引数として渡されます。送信するクエリはGETでもPOSTでも同じように扱われます。

では、以下のようなURLをWebブラウザに入力したら何が起こるでしょうか。GETリクエストで、クエリに文字列を渡しています。まるで、フォームの一部が書き換えられてしまったように見えるはずです。

```
http://127.0.0.1:8000/?name=%22/%3E-abcde-
```

図01　Webブラウザ上に任意の文字列が表示されてしまう

ソースコードに埋め込まれたテンプレート文字列を見返すと、**value="${name}"**というようになっています。つまり、クエリの「name」の文字列が**value="～"**の内部に埋め込まれるわけです。

GETのクエリ「name=%22/%3E-abcde-」の部分は、**URLエンコード**という手法で文字列が変換されています。もともとは「/>-abcde-」という文字列でした。この文字列が埋め込まれることで、HTML的に見ると**inputエレメントが閉じられて、その後の文字列がエレメントの外にある**ことになるわけです。

エレメントのvalueアトリビュートに文字列を埋め込むこの機能は、もともとフォームの入力に誤りがあったときなどに、前回入力された値をフォームに埋め込むために作った機能です。しかし、このように本来の目的を超えた、想定外の利用方法が存在するわけです。このフォームが企業のWebサイトに設置されていて、GETのパラメータとして、悪意のある内容を持った文字列が指定されたらどうなるでしょうか。この機能が、ある種の攻撃に利用されてしまうかもしれません。

このようなセキュリティホールは**クロスサイトスクリプティング**（XSS）と呼ばれています。HTMLの文字列としての性質を利用したセキュリティホールと言えます。

15-02　セキュリティホールへの対処

Webアプリケーションを使う上で、起こりうることをすべて想定してプログラムを作れば、原理的にはセキュリティホールはなくなるはずです。しかし、実際にそのようなプログラムを作るのはとても難しいか、ほとんどの場合は不可能といってよいでしょう。

そのため、セキュリティホールを出にくくするような定石をよく知り、活用する、という対策が取られます。セキュリティホールにはいくつかのパターンがあります。そのため、取るべき対策のパターンも限られています。

先ほど例に挙げたようなセキュリティホールが発生する原因として考えられるのは、Webアプリケーションで扱う文字列を適切に扱えていない、ということです。先ほどの申し込みフォームでは、フォームのアトリビュートの中に文字列を埋め込んでいました。アトリビュートの内部には**書き込んではいけない文字列**があります。ダブルクオーテーション（"）や「>」のような文字列は書き込んではいけないことになっています。そのような文字列を直接書

き込む代わりに、**実体参照文字列**と呼ばれる文字列を使って置換します。つまり、この例で示したセキュリティホールが発生した原因は、そのような「約束事」をよく理解していなかったために起こったと言えるのです。

クロスサイトスクリプティング（XSS）

Webアプリケーションのセキュリティホールとして使われるクロスサイトスクリプティングには2つの意味があるようです。Webブラウザが表示するHTMLに任意の文字列を埋め込めるという不具合自体を指して、広い意味でクロスサイトスクリプティングと呼ぶ場合があります。また、この不具合を利用して悪意のあるスクリプトを埋め込むことを指して、より狭い意味でこの言葉を使うことがあります。

スクリプトを埋め込むとはどういうことでしょうか。たとえば、先ほどのお問い合わせフォームを起動中に、Webブラウザに以下のようなURLを打ち込んでみてください。

```
http://127.0.0.1:8000/?name=%22/%3E%3Cscript%3Ealert('Hello!')%
3C/script%3E
```

このURLを使うと、HTMLの内部に**"/><script>alert('Hello!')</script>**という文字列を埋め込むことになります。結果としてHTMLにJavaScriptの実行コードが埋め込まれ、アラート表示のダイアログが表示されたはずです。

図02　アラートボックス

このような方法を使うと、より攻撃的で悪意のあるコードを埋め込むこともできます。たとえば、JavaScriptを使うとCookieの内容を取得することができます。フォーム認証では認証用のデータがCookieに保存されていることがある、ということを思い出してください。このセキュリティホールを使って、どのようなことが起こるかを想像してみてください。

クロスサイトスクリプティングの対策

クロスサイトスクリプティングを防ぐために取られる一般的な対策は、まず**HTMLのエレメントのアトリビュート値をクオーテーションで囲む**ということです。また、**動的に置き換える文字列をエスケープする**、という対策も有効です。動的に埋め込まれるダブルクオーテーション（"）や不等号がクロスサイトスクリプティングの引き金になります。これらの文字列を**実体参照文字列**で置き換えることで、JavaScriptなどが埋め込まれる危険性がほぼなくなります。実体参照文字列とは、HTMLやXMLで、特定の文字列を直接表記する代わりに利用する文字列のことです。たとえば「>」に対応する実体参照文字列は「<」、「"」は「"」というように対応が決まっています。

実体参照への変換は、基本的にHTML文字列に動的な文字列を埋め込む前に1回だけ行います。何回も変換を繰り返すと、実体参照文字列自体に含まれる「&」がさらに実体参照文字列に変換されてしまいます。そうなると、実体参照文字列から元の文字列への変換ができなくなってしまいます。

CGIのような手法でレスポンスを生成している場合は、レスポンスに埋め込む文字列の扱いによく注意する必要があります。必要な文字列を実体参照文字列に変換を行う関数を用意しておき、文字列の動的置き換えをする直前に、1回だけ変換を行うようにします。

クロスサイトスクリプティングとテンプレートエンジン

リクエスト文字列の生成にテンプレートエンジンを使う場合では、テンプレートエンジン自体が実体参照文字列への変換を行うべきです。テンプレートエンジンでは、動的な置換を行う場所に特別な記法で文字列を埋め込みます。テンプレートエンジンがこの記法を使って文字列を置換するときに、自動的に実体参照文字列への変換を行えば、クロスサイトスクリプティングは起こらずに済みます。

ただし、場合によってはHTMLのエレメント（タグ）を含む文字列を無変換で動的に置換したい場合もあります。たとえばウィジェットをテンプレート

エンジンに埋め込むときにはHTMLのタグを含んだ文字列を動的に置換します。もしウィジェットの出力するHTMLまで実体参照に変換されてしまったら、ウィジェットのフォームが正しく表示されません。このようなことを避けるため、実体参照への変換を行う置換と、行わない置換の2種類を用意することが多いようです。

既存のテンプレートエンジンを使う際、文字列置換時に実体変換が行われない場合は、扱いに注意が必要です。

ところで、本書で作ったテンプレートエンジン（SimpleTemplate）では、文字列の置き換え時に実体参照への置換を行っていません。置換時に実体参照への変換を行うようにする方法については、後ほど議論することにします。

SQLインジェクション

クロスサイトスクリプティングは、HTMLの文字列としての特性をついたセキュリティホールでした。同じように、SQLの文字列としての特性をついたセキュリティホールがあります。それが**SQLインジェクション**と呼ばれているセキュリティホールです。

HTMLと同じく、データベースを操作するために利用するSQLも文字列から構成されています。多くのWebアプリケーションでは、プログラムを効率的に作るためにSQL文字列をプログラムで動的に組み立て、データベースとの通信を行っています。たとえば、SQLを使ってデータを検索する条件、フォームに入力されたデータをデータベースに反映するためのSQLなどが動的に作られています。

具体的な例を挙げて説明しましょう。たとえば、Webアプリケーションの利用者のデータがデータベースに登録されているとします。ユーザIDから、名前を調べるためには以下のようなSQLを使います。

```
SELECT name FROM userdata WHERE userid='XXXX';
```

検索フォームを使って特定のユーザのデータを検索する場合を考えましょう。検索フォームにはユーザIDを入力します。フォームに入力した文字列を、「XXXX」の部分に埋め込んでSQLを作ります。

このとき、フォームに「' OR 's'='s」という文字列を入力したらどうなるでしょうか。最終的に以下のようなSQLが生成されることになります。

```
SELECT name FROM userdata WHERE userid='' OR 's'='s';
```

WHERE句以下は検索条件を指定している部分です。ここに**useridが''か、または's'='s'**かという条件が指定されていることになります。この条件はすべてのデータに対して「真」となりますので、結果としてデータベース上にあるすべてのユーザの名前が取り出されることになります。

このように、SQLで使われる区切り文字列などを悪用して、意図されていな命令をデータベースに送ることができるのがSQLインジェクションと呼ばれるセキュリティホールの正体です。この例のように検索条件の変更だけでなく、場合によっては、パスワードのような情報を取り出したり、データを消してしまえることもあります。

SQLインジェクションの対策

クロスサイトスクリプティングと同じく、SQLインジェクションへの対策も**動的に埋め込まれる文字列の適切なエスケープ**がベストアンサーです。クオーテーションやセミコロン（;）など、SQL文字列の中で特別な意味を持つ文字列をエスケープしてから、SQL文字列に埋め込むようにします。

PythonのDBAPIでは、**プレースホルダ**を使うとこの置換を自動的に行ってくれます。SQL文字列を動的に組み立てるときには、文字列フォーマットの機能などは使わず、極力プレースホルダを使うとよいでしょう。

また、ほとんどの既成のO/Rマッパーでは、動的に埋め込まれる文字列を適切に扱ってくれます。セキュリティに配慮して作られている既成のO/Rマッパーを使うことでも、SQLインジェクションのリスクを最小限度に抑えることができます。

クエリに関連するセキュリティホール

クロスサイトスクリプティングもSQLインジェクションも、対策を怠ると大きな被害をもたらしうるセキュリティホールといえます。ただし、テンプレートエンジンやO/Rマッパーといった開発手法を正しく使えば、適切に対

処することができるセキュリティホールです。

その他、Webアプリケーションではクエリに関連してセキュリティホールが発生する場合があります。たとえば、フォームからのPOSTリクエストを受け付ける関数などがあるとします。フォームには性別を選ぶメニューがあり、「男性」、「女性」、「無回答」の3つの項目しかないとしましょう。

フォームでは他の値を選ぶことができないのですから、POSTリクエストを受ける関数では値のチェックが必要ないと思うかもしれません。しかし実際には、GETを使ってメニューにないクエリを作ることができますし、フォームから送信されたリクエストになりすましたリクエストを、Pythonを使って作り上げることもできます。

インターネットに公開し、不特定多数のユーザに使ってもらうことを前提として作るWebアプリケーションでは、クエリの値チェックを特に厳密に行う必要があります。必ずフォームからリクエストが送られてくることを前提にしたコードには、いろいろな危険性が潜んでいる可能性が高いと言えます。値のチェックを手軽に行いたいなら、バリデータのような仕組みを使うとよいでしょう。

CHAPTER 16

RSSリーダーを作る その3

これまで作ってきたテンプレートエンジンやO/Rマッパー、ウィジェット、Webアプリケーションサーバや認証の仕組みを使って、RSSリーダーの機能拡張をしてみましょう。

16-01 拡張版RSSリーダーの仕様を決める

本書でこれまで作ってきたテンプレートエンジン、O/Rマッパー、ウィジェット、Webアプリケーションサーバは、今日的なWebアプリケーションの開発でよく使われます。本書の後半で作ってきたこのような仕組みは、Webアプリケーションの開発効率を高め、より手軽に、より安全に開発をする上で欠くことのできない仕組みとなっています。ここでは、このような仕組みを使い、RSSリーダーを再実装して、拡張してみましょう。

まずは、拡張版のRSSリーダーの仕様を簡単に決めましょう。

CSSを使ったデザインについては、前回の拡張でほぼ固まっています。テンプレートやCSSは、以前作ったものをそのまま流用します。

データ入力の遷移についても、大枠は前回の拡張で固まっています。今回は、ウィジェットとバリデータを使って、データ入力のフォーム遷移をよりスマートに実装します。

また、今回はWebアプリケーションサーバを活用してRSSリーダーを実装します。それに合わせて、簡単なユーザ認証の仕組みを組み込みましょう。ユーザ名とパスワードを入力してログインして初めて、RSSリーダーが使えるようにするのです。

事前準備として、RSSリーダー用のスクリプトファイルを入れるためのフ

ォルダを作ります。ここでは「rssreader3」としておきます。そこに、これまで作ってきたWebアプリケーション用の部品となるモジュールをコピーします。再度確認する意味を込めて、これまで作ってきたモジュールをリストアップしてみましょう。

表：使用するモジュール

ファイル名	解説
httphander.py	リクエスト、レスポンスをPythonのインスタンスとして扱えるようにするクラスを定義したモジュールです。
simpletemplate.py	テンプレートエンジンとして使うクラスを定義したモジュールです。
simplemapper.py	O/Rマッパーとして使う基底クラスを定義したモジュールです。
validators.py	フォームに入力された文字列の妥当性チェックを行うバリデータとして使うクラスを定義したモジュールです。
widgets.py	ウィジェットを定義したモジュールです。Webアプリケーションのデータ入力に利用するフォームを定義するために利用します。
simpleappserver.py	Webアプリケーションサーバとして利用するクラスを定義したモジュールです。
authentication.py	フォーム認証を行うために必要な関数やデコレータ関数一式を定義したモジュールです。
rssparser.py	RSSを読み込み、Pythonのオブジェクトに変換する関数を定義したモジュールです。

既存モジュールの変更

widgetsモジュールとauthenticationモジュールは、このRSSリーダー用に改良を加えます。まずwidgetsモジュールには、次のHiddenクラスを追加します。

List01 Hiddenクラス（widgets.py）

```
                              ⋮
class Hidden(Submit):
    """
    Hiddenフィールド用のウィジェット
    """

    def get_form(self, value=''):
        body=("""<input type="hidden" name="${name}" """
              """value="${value}" ${attrs} />""")
        t=SimpleTemplate(body)
        return t.render({'name':self.name, 'value':value,
                         'attrs':self.attrs})
                              ⋮
```

そしてauthenticationモジュールは、ログインフォームをテンプレートで表示するように修正します。主な変更部分は以下のとおりです。詳しくは本書のサポートサイトからダウンロードできるサンプルファイルを参照してください。

List02 authentication.pyの変更

```
                              ⋮
def relativepath(p):
    return path.join(path.dirname(__file__), p)

@expose
def login_form(_request, values={}, errors={}):
    res=Response()
    t=SimpleTemplate(file_path=relativepath('form.html'))
    values['password']=''
    body=t.render({'form':loginform, 'values':values,
                   'errors':errors, 'message':u'ログイン'})
    res.set_body(body)
    return res
                              ⋮
```

16-02 RSSリーダーの作成

データを保存するための仕組みと、フォームを使ったデータ入力の遷移を作るための下準備をします。データを保存するためにはO/Rマッパー用のクラスを作ります。フォームを使ったデータ入力の遷移を作るためには、ウィジェットを定義します。Webアプリケーションのロジックに直接関係のないクラスは、便利のために独立した「rssclasses.py」というモジュールファイルとして定義することにしましょう。

まずは、O/Rマッパーのクラス部分を定義します。コードの内容は、前回作ったO/Rマッパークラスの定義とまったく同じです。

List03 Rssurlクラス(rssclasses.py)

```
#!/usr/bin/env python
# coding: utf-8

import sqlite3
from os import path
from simplemapper import BaseMapper

class Rssurl(BaseMapper):
    rows=(('title', 'text'), ('url', 'text'))

p=path.join(path.dirname(__file__), 'urls.dat')
con=sqlite3.connect(p)
BaseMapper.setconnection(con)

Rssurl.createtable(ignore_error=True)
                          ⋮
```

次に、RSS巡回用URLを編集するためのフォームを定義します。フォームはウィジェットとして定義します。新規登録用、編集用に2つのフォームが必要になります。

データの新規登録フォームと編集フォームはとても似通っています。編集フォーム(editforms)と追加フォーム(addforms)の違いは一点だけです。編集用のウィジェットには、編集対象となるデータを指すために**ID**というフィールドが追加されています。

前回作ったRSSリーダーでは、編集フォームと追加フォームで、フォームのHTMLとバリデーションチェック用のソースコードが重複してしまっていました。実際、前回のRSSリーダーのソースコードでは、ソースコードが約15行、フォームのHTMLが約20行重複しています。

ウィジェットを使うことによって、HTMLの重複がなくなります。また、バリデーションチェックの処理がバリデータとして抽象化されることで、スッキリ記述できるようになっているのが分かります。

List04 フォーム用ウィジェット(rssclasses.py)

```
                          ⋮
from validators import NotEmpty, IntValidator, URLValidator
from widgets import Hidden, Text, TextArea, Submit, Reset, Form

editforms=(Text('title', u'タイトル',
             validators=(NotEmpty(),), attrs={'size':40}),
           Text('url', u'RSSのURL',
             validators=(URLValidator(),), attrs={'size':40}),
           Hidden('item_id', u'ID',
             validators=(IntValidator(),) ),
           Submit('submit', u'登録'))

editform=Form(editforms, {'action':'/edit', 'method':'POST'})

addforms=(Text('title', u'タイトル',
             validators=(NotEmpty(),), attrs={'size':40}),
          Text('url', u'RSSのURL',
             validators=(URLValidator(),), attrs={'size':40}),
          Submit('submit', u'登録'))

addform=Form(addforms, {'action':'/add', 'method':'POST'})
```

追加フォームを作る

次にRSSリーダーのアプリケーション本体を作成します。これは「rssreader3.py」というファイル名にしましょう。

まず実際にリクエストを受け取る関数を作ります。今回はWebアプリケーションの開発にWebアプリケーションサーバを活用します。本書で作ったWebアプリケーションサーバを使うには、リクエストを受け取る関数を定義する必要があります。定義した関数にデコレータを使って加工を施すと、リクエストを受け取れるようになります。

まずは、データを追加するためのフォームを作りましょう。バリデーションチェックなどを含めた遷移を実装します。

データの追加を行う遷移一式を作るためには、**フォームを表示する関数**と**フォームのデータを受け取り登録を行う関数**の2つを作ります。フォームの表示にはウィジェットを作ります。また、フォームからデータを受け取る関数では、バリデータを使ってデータの妥当性チェックを行います。必要があればフォームの再表示をして、正しいデータを入力するようユーザに促します。

以下が登録の遷移を実現する2つの関数です。前半に必要なモジュールを

インポートする部分があります。その後、リクエストを受ける関数が並んでいます。

add_form()関数でフォームを表示し、add()関数で妥当性のチェックやデータの登録を行います。リクエストのクエリは関数の引数となって受け取ることができます。Webアプリケーションを開発するときには、ごく普通の関数を作るときと同じような手法が使えるわけです。なお、この関数には、**http://127.0.0.1:8000/add_urs**のようなURLでアクセスします。また、secured_exposeデコレータを使って、ログインをしないと関数にアクセスできないように指定しています。

List05 新規追加フォーム（rssreader3.py）

```
#!/usr/bin/env python
# coding: utf-8

from os import path
from copy import copy
from simpleappserver import expose, test
from httphandler import Response, get_htmltemplate
from simpletemplate import SimpleTemplate

from authentication import secured_expose, relativepath, checklogin

from rssparser import parse_rss
from rssclasses import Rssurl, addform, editform

@secured_expose(checkfunc=checklogin)
def add_form(_request, values={}, errors={}):
    res=Response()
    t=SimpleTemplate(file_path=relativepath('form.html'))
    body=t.render({'message': u'RSS巡回用URLの追加',
                   'form':addform,
                   'values':values, 'errors':errors})
    res.set_body(body)
    return res

@secured_expose(checkfunc=checklogin)
def add(_request, title='', url=''):
    res=Response()
    values, errors=addform.validate({'title':title, 'url':url})
    if [ x for x in Rssurl.select(url=url)]:
        errors['url']=u'このURLは登録済みです'
    if errors:
        return add_form(_request, values, errors)
    Rssurl(title=title, url=url)
    t=SimpleTemplate(file_path=relativepath('posted.html'))
    body=t.render({'message': u'巡回用URLを追加しました'})
    res.set_body(body)
    return res
                    ⋮
```

編集フォームを作る

追加フォームができたので、今度はデータを編集するためのフォームを作ります。追加フォームと同じように、遷移を作るには2つの関数が必要です。1つはフォームを表示する関数edit_form()、もう1つはデータの更新をする関数edit()です。

追加用の関数と同様に、ウィジェットを使ってフォームを表示し、バリデータで妥当性のチェックをし、期待通りのデータが登録されていれば、データを登録します。

追加用の関数と異なっている点は、編集対象となるデータを指定する引数IDが追加されている点です。edit()関数では、引数idを使って編集対象となるデータを特定し、O/Rマッパーのクラスを使ってインスタンスを取得しています。インスタンスのアトリビュートに代入後、update()メソッドを呼ぶことでデータの更新を行います。

List06 編集フォーム（rssreader3.py）

```
                        ⋮
@secured_expose(checkfunc=checklogin)
def edit_form(_request, item_id, values={}, errors={}):
    res=Response()
    t=SimpleTemplate(file_path=relativepath('form.html'))
    if not values:
        for item in Rssurl.select(id=item_id):
            pass
        values={'item_id':item_id, 'title':item.title,
'url':item.url}
```

```
    body=t.render({'message': u'RSS巡回用URLの編集',
                   'form':editform,
                   'values':values, 'errors':errors})
    res.set_body(body)
    return res

@secured_expose(checkfunc=checklogin)
def edit(_request, item_id, title='', url=''):
    res=Response()
    values, errors=editform.validate({'item_id':item_id,
                            'title':title, 'url':url})
    if errors:
        return edit_form(_request, item_id, values, errors)
    for item in Rssurl.select(id=item_id):
        item.title=title
        item.url=url
    t=SimpleTemplate(file_path=relativepath('posted.html'))
    body=t.render({'message': u'巡回用URLを編集しました'})
    res.set_body(body)
    return res
                         ⋮
```

編集用URL一覧ページを作る

新規追加、編集用のフォームができました。次に編集フォームにリンクするリストを作ります。

リストを表示するためには、O/Rマッパーのクラスを使い、RSS巡回用URLオブジェクトをすべて取得します。取得したオブジェクトをリストに格納し、テンプレートエンジンに渡して表示をする、という手順でリストを表示します。テンプレートエンジンとO/Rマッパーのおかげで、コードの構成がとてもシンプルになっています。また、Webアプリケーションサーバを使って開発しているため、関数を定義するだけでレスポンスに対応することができます。

List07 編集フォームにリンクするリスト(rssreader3.py)

```
                         ⋮
@secured_expose(checkfunc=checklogin)
def listurl(_request):
    res=Response()
    rsslist=Rssurl.select()
    t=SimpleTemplate(file_path=relativepath('urllist.html'))
    body=t.render({'rsslist': rsslist})
    res.set_body(body)
    return res
                         ⋮
```

RSS一覧ページを作る

最後に、登録されているRSSを巡回し、一覧ページを表示するための関数を作ります。まずはO/Rマッパーのクラスを使って巡回用のURLを取得し、rssparserモジュールのparse_rss()関数を使ってRSSをPythonのオブジェクトに変換します。

変換した結果は辞書としてリストに格納します。辞書をリストに格納し、テンプレートエンジンに渡してRSSを表示します。

List08 RSS一覧ページ編集フォームにリンクするリスト

```
                         ⋮
@secured_expose(checkfunc=checklogin)
def index(_request):
    rsslist=[]
    try:
        for rss in Rssurl.select(order_by='id'):
            rsslist.extend(parse_rss(rss.url))
    except:
        pass

    res=Response()
    p=path.join(path.dirname(__file__), 'rsslist.html')
    t=SimpleTemplate(file_path=p)
    body=t.render({'rsslist':rsslist[:20]})
    res.set_body(body)
    return res
```

これでRSSリーダーは完成です。

テンプレートの作成

最後にテンプレートエンジンが使用するテンプレートを作成します。作成するのは次の4つです。<body>要素のみ紹介しますので、詳しくは本書のサポートサイトからダウンロードできるサンプルファイルを参照してください。

List09 form.html

```
<body>
<h1 class="header">${message}</h1>
${form.display(values, errors)}
</body>
```

List10 posted.html

```
<body>
<h1 class="header">簡易RSSリーダー</h1>
<p class="description">${message}</p>
<a href="./listurl">RSSのリストに戻る</a>
</body>
```

List11 frsslist.html

```
<body>
<h1 class="header">簡易RSSリーダー</h1>
<h2 class="title">RSSの閲覧</h2>
<div class="control">
    <a href="listurl">RSSの編集</a>
    <a href="add_form">新規追加</a>
</div>
<ul>
$for item in rsslist:
    <li>
        <dt>
            <a href="${item.get('link', '')}">
                ${item.get('title', '')}
            </a>
            (${item.get('pubDate', '')})
        </dt>
        <dd>
            ${item.get('description', '')}
        </dd>
    </li>
  $endfor
</ul>
</body>
```

List12 urllist.html

```
<body>
<h1 class="header">簡易RSSリーダー</h1>
<h2 class="title">RSSの追加，編集</h2>
<div class="control">
    <a href="/add_form">新しいRSSを追加</a>
    <a href="/">RSS一覧に戻る</a>
</div>
<ol>
$for item in rsslist:
    <li>
        ${item.title}
        <span class="control">
        <a href="./edit_form?item_id=${item.id}">編集</a>
        </span>
    </li>
$endfor
</ol>
</body>
```

RSSリーダーの実行

このRSSリーダー・Webアプリケーションを実行してみましょう。すでにPythonのWebサーバが起動している場合は終了させてから、「rssreader3.py」を起動します。そして**http://127.0.0.1:8000/**にアクセスします。

ログインフォームが表示されますので、ユーザ名（user）とパスワード（pass）を入力すると、RSSの一覧が画面が表示されます。操作方法自体はrssreader2.pyと同じです。

図01　ログイン画面

16-03　機能拡張のためのヒント

RSSリーダーの機能はほぼ完成しました。最初に作ったRSSリーダーはフォームが1つだけあるとても単純なものでした。ページを進めるにつれ、少しずつより新しい開発手法を取り入れ、機能拡張をしてきました。新しい機能を実装しつつ、コードの量はあまり増えていません。処理の内容を抽象化し、共通部分を抜き出すことで、開発を効率化してきたおかげと言えます

ほぼ完成しているRSSリーダーも、まだ機能追加の余地があります。読者の皆さん自身の勉強として、以下のような機能追加をしてみてはいかがでしょうか。

ユーザ登録機能

ユーザ名とパスワードをデータベースに保存し、データベースの内容を使ってユーザ認証を行う機能を追加するとRSSリーダーはずっと使いやすくなるはずです。ユーザの認証はauthentication.pyのchecklogin()という関数で行っています。この関数を書き換え、データベースの内容を参照するようにすることになります。また、ユーザ登録、内容を編集するためのフォームと遷移一式が必要になります。

RSSの内容をデータベースに登録する

現状のRSSリーダーでは、RSSを表示しようとするたびにRSSを読み込んでいます。このような仕様だと、RSSの読み込みに時間がかかりますし、RSSを配信しているサーバに余計な付加をかけることになります。読み込んだ内容をデータベースに登録しておき、一定時間経つと再度RSSを読みに行く、という実装の方がスマートと言えます。

このような機能を実装するためには、まずRSSのデータを登録するような仕組みを作る必要があります。RSS表示時にはデータベースに登録された内容を利用します。また、最後にRSSを読み込んだ時間をどこかに保存しておくとよいでしょう。

APPENDIX

APPENDIX

Webアプリケーションフレームワークの活用

最近のWebアプリケーションの開発では、**Webアプリケーションフレームワーク**と呼ばれるものが利用されることが多くなってきました。本書の後半で作ったような、テンプレートエンジンやO/Rマッパーというようなライブラリなどを組み合わせ、より扱いやすくしたものをWebアプリケーションフレームワークと呼びます。ここでは、Pythonで利用できるWebアプリケーションフレームワークについて、簡単に解説したいと思います。

A-01 Webアプリケーションフレームワークとは

Webアプリケーションの開発に利用されるテンプレートエンジン、O/Rマッパーのような部品を集め、Webアプリケーションの開発により利用しやすいようにしたものをWebアプリケーションフレームワークと呼びます。フレームワークとは「枠組み」という意味の英語です。Webアプリケーションの開発に便利な枠組みを提供するためのライブラリ集、というような意味と捕らえてください。Webアプリケーションフレームワークは、省略されてフレームワークと呼ばれることがあります。本書でも、これからはフレームワークと呼ぶことにしたいと思います。

近年のWebアプリケーション開発では、フレームワークが利用されることが多くなってきています。フレームワークを使うと、高機能で複雑なWebアプリケーションを手軽に開発できるからです。フレームワークの利用は、Webアプリケーション開発の世界におけるトレンドといってもよいかもしれません。

PythonとWebアプリケーションフレームワーク

Pythonでは比較的以前から、テンプレートエンジンやアプリケーションサーバのようなWeb開発に利用できるライブラリが多く存在していました。そのようなライブラリの多くは、無料で利用できます。Webアプリケーションの開発者は、自分の気に入ったライブラリを手に入れて、組み合わせて利用していました。ライブラリの組み合わせにはたくさんの種類があり、開発者の好みもいろいろだったので、その方が都合がよかったのでしょう。

また、Pythonの世界では古くから「Zope」というフレームワークがよく利用されていました。そのようなこともあり、Zope以外の特定のフレームワークを多くの開発者が使う、という状況が生まれづらい、というような状況があったようです。

しかし、ここ最近は状況が変わりつつあります。Zope以外にも、いくつかの主力と呼べるフレームワークが登場してきたのです。Zopeは歴史のあるフレームワークですが、半面重厚すぎて使いづらい面を持っていました。より軽量なフレームワークが求められていた、という状況も、新世代のフレームワーク登場に一役買っていたのかもしれません。

ここでは、Pythonの代表的なフレームワークとして「Django」、「Turbo Gears」、そしてZopeをベースに作られている「Plone」の3つを特に取り上げて、簡単に特徴を解説したいと思います。また、PythonのWebアプリケーション実装を抽象化する試みとしてWSGIを取り上げます。

Webアプリケーションフレームワークの分類

実際にフレームワークの特徴を解説する前に、フレームワークの分類方法について簡単に触れてみたいと思います。さまざまな種類のあるフレームワークですが、どんなフレームワークも持っている機能が何種類かあります。この機能を元に、フレームワークを分類することが可能なのです。

リクエストハンドリングの方法

フレームワークには、Webブラウザのようなクライアントから、Webアプリケーションを構成するプログラムにリクエストを受け渡す方法が備わっています。この手法によって、フレームワークを分類することができます。

大まかに分けて、リクエストに応答するための関数を作るタイプのフレームワークと、クラスを作るタイプのフレームワークの2種類があります。

データ永続化(保存)の方法

Webアプリケーションを作るときに、データの永続化処理は必須の機能です。多くのフレームワークは、データ保存を手軽に行うための方法を持っています。O/Rマッパーを使ってリレーショナルデータベースとの接続性を高める手法が主流のようです。また、独自のデータベースをフレームワーク自体が内蔵している場合もあります。

テンプレートエンジンと記法

複雑なフォームや遷移、綺麗なデザインのWebアプリケーションを作るためにテンプレートエンジンは必須の機能です。多くのフレームワークではテンプレートエンジンを内蔵しています。

プログラムから表示に必要なデータを与え、動的に置き換える部分に特殊な記法を使った文字列を埋め込む、という基本的な動きはどのテンプレートエンジンにも共通した機能です。ただし、記法や文法などにはテンプレートエンジンによっていろいろな種類があります。この点を使って、テンプレートエンジンを分類することができます。

フォームの生成方法と遷移のコントロール

Webアプリケーションではいろいろな種類の入力フォームを作ります。たいていのWebアプリケーションでは、入力フォームを手軽に作るための仕組みを備えています。また、エラーハンドリングやバリデーションチェックの仕組みも備わっています。このような仕組みを元に、フレームワークを分類することができます。

A-02 Django

DjangoはPython製のオープンソースWebアプリケーションフレームワークです。もともと新聞社のサイトを運営するために作られた内製のフレームワークでしたが、2005年に**BSDライセンス**でオープンソース化されました。実際に付加の高いサイトで利用されていた実績のあるフレームワークであることや、洗練された開発スタイルなどが好評を呼び、世界中でPythonの開発者たちの注目を集めています。国外だけでなく、日本のコミュニティ活動も活発で、定期的に勉強会が開催されています。

Django
http://www.djangoproject.com/

Djangoは、Pythonが動く環境であればほぼどんな環境でも稼働します。Linuxだけでなく、Mac OS XやWindowsなどでDjangoを動かすことができます。

Djangoのリクエストハンドリング

Djangoでは、**urls**と呼ばれる正規表現のセットを登録することでリクエストのハンドリングを行います。まず、正規表現とリクエストを受け取る関数をペアにしてセットしておきます。Webサーバの受け取ったリクエストのパスを評価し、urlsに登録された正規表現と照らし合わせて、ヒットする正規表現が見つかったら、関数に処理を渡す、という仕組みでリクエストをハンドリングするのです。

以下のような設定をしておくと、**~/myapp/**というリクエストが「mysite.myapp.views」というモジュールにある「index()」という関数に渡されます。

```
urlpatterns = patterns('',
    (r'^myapp/$', 'mysite.myapp.views.index'),
)
```

また、urlsにはパラメータを記述することもできます。パラメータ付きのurlsの設定は以下のようになります。以下のような正規表現パターンを登録しておくと、**~/myapp/10/**というリクエストのうち「10」という数値がパラメータとして扱われます。リクエストを受け取る関数「detail()」では、**some_id**という引数としてこの数値を受け取ることができます。

```
(r'^myapp/(?P<some_id>¥d+)/$', 'mysite.myapp.views.detail')
```

このように、リクエストと関数の関係をurlsという正規表現を使って設定して、リクエストのハンドリングを行うのがDjangoの手法です。

また、Djangoには、開発用に利用する軽量のWebサーバが付属しています。このWebサーバはPythonで書かれていますので、Pythonが動く環境であれば、ApacheなどのWebサーバを特別にインストールしなくてもDjangoを使ってWebアプリケーションを開発し、動かすことができます。ただし、作ったWebアプリケーションを公開するときには、Apacheのようなより本格的なWebサーバを利用することが多いようです。

Djangoのデータ永続化の手法

Djangoでは、データを保存するためにリレーショナルデータベースを使います。データ保存時にはSQLを直接操作するのではなく、O/Rマッパーを利用します。Djangoは独自のO/Rマッパーを持っていて、これを利用します。PostgreSQLやMySQLをはじめ、多くのリレーショナルデータベースに対応しています。

DjangoのO/Rマッパーでは、テーブルを定義するためにクラスを作ります。クラスのアトリビュートとしてカラムを表現するためのクラスインスタンスを設定することで、テーブルのカラムや型を定義します。**CharField**や**DateTimeField**というクラスの型によって、カラムの型を表現しています。

以下が、O/Rマッパーを活用したクラス定義の簡単な例です。

```
from django.db import models

class ORClass(models.Model):
    data1 = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
```

O/Rマッパーのクラスには、データベース上のデータを操作するためのメソッドが自動的に登録されます。このメソッドを使うことで、データベース上のデータを操作できます。

O/Rマッパーのクラスを生成することで、データベース上にデータを登録できます。データベース上のデータにはインスタンスのアトリビュートとしてアクセスできます。ただし、変更をデータベースに反映させるためには、明示的にメソッドを呼ぶ必要があります。

Djangoのテンプレートエンジンと記法

Djangoは独自のテンプレートエンジンを持っています。HTMLに独自の記法を埋め込むことで、動的に置き換える部分を定義します。以下は、Djangoのテンプレートを使った記法の一例です。**{%～%}**や**{{~}}**で囲まれた部分が特別な意味を持っています。

```
{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Djangoのテンプレートエンジンは、変数などを埋め込んでテンプレートを動的に書き換える機能だけでなく、繰り返しや条件分岐といった機能も備えています。また、テンプレートの一部を部品化したり、Pythonのクラスのようにテンプレートの一部を継承することもできます。

Djangoのフォームの生成方法と遷移のコントロール

Djangoのフォーム生成はとてもよくできています。O/Rマッパーのクラス定義を自動的に読み込み、綺麗で使いやすいCRUDフォームと遷移を自動的に生成してくれるのです。DjangoのCRUDフォーム自動生成機能は**admin**（アドミン）と呼ばれています。

もともとDjangoはWebサイトを運用する**CMS**（コンテントマネジメントシステム）を作るためのフレームワークとして開発が始まりました。そのようなWebアプリケーションでは、サイトの管理者が使う入力フォームよりも、データを表示するテンプレートにより多くの開発時間を割きたい場面が多いはずです。そのような経緯もあり、データを登録するだけで綺麗で便利なフォームを自動的に作る機能が求められていたのでしょう。

Djangoのadminのような自動生成のフォームは便利な半面、不便な場合もあります。フォームをカスタマイズしたいときに使い回しがしづらいのです。フォームのカスタマイズをしたいときに利用するために、newformsと呼ばれる仕組みがDjangoに追加されました。

Djangoのまとめ

Djangoはアクセスの集中するWebサイトで利用されてきたという実績を持つフレームワークです。キャッシュと呼ばれる付加対策を簡単に設定できたりと、高い負荷に耐えうるサービスを開発できる素養を持った、懐の深いWeb開発アプリケーションと言えるでしょう。開発体制もとても安定していて、安心して利用できるフレームワークでもあります。newformsが取り入れられて、応用の幅がより広がってきています。

実動するサービスに利用されていて、フレームワークとして安定している半面、新しい要素を取り入れることにあまり積極的でない面もあるようです。安心感を取るか、より新しい技術を取るか、好みの分かれるところかもしれません。

A-03 TurboGears

TurboGearsは2005年に開発が始まった比較的新しいフレームワークです。もともとは、オリジナルの作者であるKevin氏がRSSリーダーを作るために作り始めたフレームワークでした。フレームワーク部分のみを公開したところ、設計の筋の良さから多くの開発者の注目を集めるようになりました。2006年にバージョン1.0が公開されています。

TurboGears
http://www.turbogears.org/

TurboGearsの特徴は、テンプレートエンジンやO/Rマッパーを入れ替えることができる、という点です。そのような機能はプラグイン形式になっているため、自分の好みや要求に合わせてブロックのように部品を入れ替えることができます。TurboGearsの開発者は新しいものを取り入れることに比較的貪欲です。

TurboGearsはWebサーバとしてPython製の**CherryPy**というアプリケーションサーバを活用します。このため、Pythonが動けばどのような環境でも稼働します。LinuxやMac OS Xだけでなく、Windowsでも利用できます。

TurboGearsのリクエストハンドリング

TurboGearsでリクエストをハンドリングする主体となるのはPythonのクラスです。**Controller**（コントローラ）と呼ばれるクラスを継承して、メソッドを定義することでリクエストをハンドリングするための仕組みを定義していきます。この仕組みは、TurboGearsで利用しているWebアプリケーションサーバのCherryPyのものをそのまま活用しています。

リクエストのルートになる**RootController**というクラスが定義されています。このクラスを継承すると、ルート以下のアクセスを受け付けることができるようになります。RootContollerを継承したクラスに「foo()」というメソッドを定義すると、**~/foo**というリクエストを受け付けることができます。このような仕組みを使って、リクエストに応答するメソッドを定義していきます。

TurboGearsのデータ永続化の手法

TurboGearsでは、O/Rマッパーを使ってリレーショナルデータベースを使ってデータの保存を行います。利用するO/Rマッパーはプラグインを使い入れ替えることができます。必要に応じていろいろなO/Rマッパーを利用できますが、標準ではSQLObject（http://www.sqlobject.org/）とSQLAlchemy（http://www.sqlalchemy.org/）という2つのO/Rマッパーが利用できます。

SQLObjectは歴史と実績のあるO/Rマッパーです。オープンソースだけでなく、商用を含めたリレーショナルデータベースに対応しているのも魅力の1つです。

SQLAlchemyは比較的新しいO/Rマッパーです。SQLObjectに比べ、柔軟で先進的な機能を持っているのが特徴です。SQLObjectが必要とされるほぼすべての機能を提供することを目指しているのに対して、SQLAlchemyは細かい機能をカスタマイズでき、いろいろなシーンに柔軟に対応できるのが特徴です。

TurboGearsのテンプレートエンジンと記法

TurboGearsではテンプレートエンジンもプラグインによって拡張可能です。標準では、Kid（http://www.kid-templating.org/）とGenshi（http://genshi.edgewall.org/）という2つのテンプレートエンジンに対応しています。

KidもGenshiもとても似た記法を持ったテンプレートエンジンです。Djangoのテンプレートエンジンと異なり、XMLやHTMLのアトリビュート領域に特別な記法を使ってロジックを記述します。

以下がKid、Genshiのテンプレートです。

```
<h1 py:content="title">タイトル</h1>
<p py:content="body">
  本文
</p>
<ul>
  <li py:repeat="item sequence">
    <span py:replace="item" />
  </li>
</ul>
```

HTMLの中に見える**py:〜**という記法をテンプレートエンジンが解釈します。一番上に見える<h1>タグの中には「py:content="title"」という記述が見えます。この記述があると、h1エレメントの内部がtitleというオブジェクトで置換されます。つまり、テンプレートエンジンで出力するときには「タイトル」という文字が置き換わるのです。また、「py:repeat」という部分は繰り返しを意味します。sequenceというオブジェクトを元に繰り返しを実行し、liエレメント自体が繰り返して出力されます。このように、HTMLやエレメントを単位に処理を繰り返していくのがKidやGenshiの記法の特徴です。

このような記法にはいくつか良い点があります。まず、テンプレートを表示する時点でHTMLやXMLとしての妥当性チェックが行われるので、タグの閉じ忘れや非対応のない出力を必ず得られる、ということです。また、動的に置き換わる部分がHTMLのタグに埋め込まれるため、HTML編集ソフトとの親和性が高い、というのも特徴の1つです。埋め込み型のテンプレートエンジンに比べて、開発者が作ったテンプレートをデザイナーに渡し、デザインを変更する、という連携を比較的とりやすい、という特徴を持っています。

半面、ロジックの多くがタグの内部に埋め込まれてしまうため、タグの内部が長くなってしまう、という欠点があります。

TurboGearsのフォームの生成方法と遷移のコントロール

TurboGearsでは、**Widgets**（ウィジェッツ）という仕組みを使ってフォームを表示します。本書で作ったウィジェットととても似た仕組みを持っています。フォームの部品をクラスインスタンスとして定義して、フォームを表示します。Djangoのような自動表示の仕組みはありませんが、代わりにとても柔軟にフォームの生成が可能です。

TurboGearsに組み込みのフォーム部品には、Ajaxを使って遷移を起こさず非同期で処理を行うものがいくつかあります。そのような部品を使ってフォームを使うと、先進的なインターフェースを持ったWebアプリケーションをとても手軽に作ることができます。また、フォームの部品は独自に拡張することもできます。

遷移や妥当性チェックを行うためには、TurboGearsに付属しているデコレータ関数を使います。ウィジェットやバリデータを組み合わせて利用し、フォームの遷移を自由に組み立てていくのがTurboGearsの流儀です。

フォーム生成の仕組みも、TurboGearsでは切り替えることができます。

TurboGearsのまとめ

Webアプリケーションで多くの部品を自由に選び、組み合わせることができるのがTurboGearsの利点です。Web 2.0的な指向を持った部品も多く公開

されています。そのような部品をインストールすることで、TurboGearsの機能を自由に拡張できます。

TurboGearsとDjangoはお互いに比較されることの多いフレームワークです。Djangoが大規模サイトを運用するためのフレームワークを指向しているのに対し、TurboGearsは中規模までのWebアプリケーションを作ることを指向したフレームワークと分類できます。

実稼働しているサイトに利用されているDjangoの開発コミュニティは比較的保守的ですが、TurboGearsの開発コミュニティは新しい技術を取り入れることや変化に積極的なように見受けられます。そのような意味でも、Djangoとは違った特徴と指向性を持ったフレームワークであると言えます。

A-04 Plone

PloneはWebアプリケーションサーバZope（http://www.zope.org/）上に構築されています。Webサイトなどを管理するために利用する**CMS**（コンテントマネジメントシステム）と紹介されることが多いのですが、内部的にはフレームワークと呼ぶにふさわしい重厚な機能を備えています。ここでは、Ploneのフレームワークとしての側面に重点を置いて解説したいと思います。

Plone
http://www.plone.org/

Ploneは、Webサーバからデータを保存するための仕組み（オブジェクトデータベース）まで、Webアプリケーションを作るための環境をすべて自前で持っています。Ploneは、いくつかのフレームワークを積み重ねるように利用して構築されています。まず、リクエストのハンドリングやデータの保存、テンプレートエンジンといった基本的な機能はZopeに負っています。また、フレームワークとしての基本的な機能はZope上で利用できる**CMF**（Content Management Framework）と呼ばれるフレームワークを活用しています。この2つがPloneを構築するためのベースとなっています。

また、2つの基本機能をより使いやすくするために、**Archetypes**というフレームワークを搭載しています。Archetypesを使うことによって、Ploneを使った高効率な開発が行えるようになります。

Ploneのリクエストハンドリング

Ploneでは、リクエストを受け付けるWebサーバとしてZopeに内蔵されたZServerを利用します。ZServerが受け付けたリクエストは、パスなどを元に分割されます。Plone内には、リクエストを受け付けるPythonのクラスインスタンスが多く登録されています。Zope内に登録されたクラスのアトリビュートの階層をたどり、ZServerが適切なオブジェクトを見つけ出し、リクエストを届けます。最終的にクラスに定義されたメソッドが呼び出され、リクエストに応答します。

Ploneのデータ永続化の手法

Plone（Zope）のデータ永続化の方法は少し特殊です。ZopeはZODBと呼ばれる独自のデータベースを搭載しています。ZODBは**オブジェクトデータベース**という種類のデータベースです。

リレーショナルデータベースのようにテーブルを作ってデータを保存するのではなく、Pythonのオブジェクトをシリアライズし、そのままデータベースに保存できます。データベースに保存したいデータがある場合は、Pythonの永続化用のベースクラスを継承したクラスを書くだけでよいのです。データが変更されたら、データベースが変更を自動的に感知し、保存の処理を行ってくれます。

Ploneのデータベースに保存されるのはPythonのオブジェクトそのものです。Ploneの中に登録されているファイル、画像などはすべて、メソッドとデータを持ったPythonのオブジェクトということになります。

Ploneのテンプレートエンジンと記法

Ploneは2種類のテンプレートエンジンを搭載しています。1つは**DTML**と呼ばれる埋め込み式の記法を持ったテンプレートエンジンです。もう1つは**Page Template**（ページテンプレート）と呼ばれるテンプレートエンジンで、TurboGearsのKidやGenshiのようにアトリビュート領域にロジックを埋め込む記法を持っています。DTMLは、現在ではごく一部でしか利用されていま

せん。Ploneで主に使われるのはPage Templateです。

Page Templateでは、HTMLやXMLのアトリビュート領域にロジックを埋め込みます。以下がPage Templateの記法を使ったテンプレートのサンプルです。

```
<h1 tal:content="title">タイトル</title>
<p tal:content="body">
  本文
</p>
<ul>
  <li tal:repeat="item sequence">
    <span tal:replace="item" />
  </li>
</ul>
```

TurboGearsのKidやGenshiと同様に、アトリビュート領域に特殊な表記を埋め込む形でテンプレートを記述していき、エレメント単位で処理が実行されます。エレメントの置換や繰り返し、条件分岐などKidやGenshiが持つ機能の他、より高度な機能も備えています。全体で利用するマスターテンプレートを定義する機能や、テンプレートを部品化したり、継承する機能などを備えています。

Ploneのフォームの生成方法と遷移のコントロール

Ploneでは、フレームワークの基本的な機能を提供するZopeとCMFに加え、Archetypesというフレームワークを利用して開発を行います。Archetypesでは、データを保存するための領域、CRUDフォームを作るための定義や妥当性チェックを含めた設定を**スキーマ**と**ウィジェット**という形で抽象化しています。スキーマもウィジェットもPythonのクラスで、データ型や表示したいフォームの種類、適用したい妥当性チェックの種類によってクラスを使い分け、定義します。Ploneはとても豊富なスキーマやウィジェットを標準で持っています。数値や文字列、日付といった基本的なデータ型だけでなく、画像のようなオブジェクトもデータとして登録することができます。また、標準のCRUDフォームを使う限り、スキーマとウィジェットの定義をするだけで自動的にCRUDフォームを表示する仕組みを持っています。

Plone 3.0というバージョンからは、ウィジェットやスキーマを定義するだけで、Ajax風のクールなインターフェースを自動的に表示します。

Ploneのまとめ

Ploneは表面的にはアウト・オブ・ザ・ボックスとしてすぐに使えるCMS、または情報共有ポータルWebアプリケーションとして作られています。そのような、Ploneの外部に見える機能は内部のフレームワークを利用して作られています。Ploneの持つ高度な機能は、Ploneが内蔵するフレームワークの高機能ぶりを証明していると言えると思います。

DjangoやTurboGearsに比べると、Ploneはとても重厚なフレームワークです。Webアプリケーション開発のかなりの部分をカバーできる機能を持っています。開発コミュニティもとても活発で、バージョンアップのたびに新しい機能を取り込み、機能が充実してきています。

重厚で高機能なフレームワークである半面、フレームワークとしてのPloneを理解するためには多少の知識と時間を要するかもしれません。ただし、いったん覚えてしまえばとても高い効率でWebアプリケーションを開発できるようになります。CMSのように頻繁な仕様変更が要求されるタイプのWebアプリケーションを、すばやい速度で開発するようなスタイルの開発に向いたフレームワークだと言えます。

A-05 WSGI

Pythonにはとても多くのWebアプリケーションフレームワークが存在します。それぞれフレームワークが独自の作りを持っているため、たとえばあるフレームワークで作った関数などを他のフレームワークで利用するためには、ほとんどすべて書き換える必要がありました。このように、Webアプリケーションを作る際に**コードの再利用がほとんどできない**ということを以前から問題に思う開発者が少なくありませんでした。

Webアプリケーションを作る際、コードの再利用性を高めるために標準化の試みが何度か行われてきました。これから紹介するWSGIは、その中でも最も有力で将来性のある標準化の試みといえます。WSGIはそのままアルフ

ァベット読みをされる場合と、「ウィスキー」と呼ばれる場合があるようです。

WSGIとは

Pythonで作るWebアプリケーションの実装方法の**標準化仕様**です。Python 2.5からは、WSGIのリファレンス実装**wsgiref**というモジュールが標準ライブラリに登録されています。このことからも、WSGIがPythonのWebアプリケーション実装標準化手法の最右翼であることが分かると思います。

WSGIには、PythonでWebアプリケーションを作る際の作法が定義されています。サーバ、アプリケーション、ミドルウエアといったように、何を実装するかによって標準化の手法がいくつかの部分に分かれています。たとえば、Webサーバの開発者はサーバ実装の手法に沿って開発を行います。フレームワークの実装者はアプリケーション実装の手法に、テンプレートエンジンやキャッシュなどの実装者はミドルウエア実装の手法に沿う、という形で実装を進めます。

たとえば、WSGIに準拠したフレームワークはWSGIに準拠したWebサーバを自由に取り替え、選ぶことができます。このように、WSGIの手法を活用してWebアプリケーションを構成する部品を実装することで、アプリケーションの再利用性を高めることができるのです。

WSGIの仕様は、Pythonの言語やライブラリの機能拡張要望などを記述したPEP（Python Enhancement Proposals）の333番目（http://www.python.org/dev/peps/pep-0333/）に詳しく記載されています。

WSGIアプリケーション

本書で繰り返し解説してきたように、Webアプリケーションの動作原理はとても単純です。**リクエストを受け、適切なレスポンスを返す**、ということだけです。WSGIのうち、Webアプリケーション部分の標準仕様を定義したWSGIアプリケーションもまた、とても単純な仕組みで構成されています。

WSGIアプリケーションとして要求されているのは、以下の2つだけです。

- **2つの引数を受け付ける呼び出し可能オブジェクトであること**
- **レスポンスをリストやタプルのようなシーケンス、またはイテレータとして返すこと**

以下のソースコードは、PEP 333に掲載されているWSGIアプリケーションのサンプルコードです。

```
def simple_app(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type','text/plain')]
    start_response(status, response_headers)
    return ['Hello world!¥n']
```

environという引数には、CGIに渡される環境変数と同じ情報が辞書型のオブジェクトに代入されます。**start_response**は関数など呼び出し可能オブジェクトが代入されています。この呼び出し可能オブジェクトはWSGI準拠のサーバ側で定義されていて、ステータス行とヘッダを受け取ってレスポンスを返すための準備をします。

関数simple_app()の内部では、ステータスとヘッダを指定してstart_response()関数を呼び出しています。その後、リストの形式でレスポンスの本文を返しています。最小限度のWSGI準拠のWebアプリケーションは、これだけで作れてしまいます。なお、WSGIアプリケーションは呼び出し可能オブジェクトであればよいので、クラスとして実装することもできます。

実際のWSGIアプリケーションでは、URLのパスによって適切な関数に処理を振り分けたり、エラー処理をする、というような機能を実装しなければなりません。実際に開発に利用するWSGIアプリケーションは、もっと複雑になるはずです。

WSGIミドルウエア

WSGIの定義には、サーバ、アプリケーションだけでなく、**ミドルウエア**という仕様も定義されています。Webアプリケーションで利用するプログラムのように、実際の処理を定義するためのコードでなく、もっと抽象的に利用する機能の再利用性を高めるための仕様です。たとえば、テンプレートエン

ジンや、レスポンスの内容をメモリなどに一時的にコピーしておき、Webアプリケーションの動作速度を高める目的で利用するキャッシュなどがミドルウエアとして実装できます。

WSGIミドルウエアは、WSGIアプリケーションが呼ばれるのに先立って呼ばれます。事前に環境変数を操作したり新しいデータを追加するなどして、特定の処理を行います。または、WSGIアプリケーションの返すレスポンスを処理して、レスポンスを改めて作り直す、というような働きをすることもあります。

これまで、テンプレートエンジンやキャッシュなどは、フレームワークごとに作る必要がありました。WSGIミドルウエアの仕様に準拠することによって、より再利用性の高い仕組みが作れるようになるわけです。

WSGIの今後

WSGIは、PythonのWeb開発の世界で大きなトレンドになっています。DjangoやTurboGearsなどは、どちらもWSGIへの準拠を宣言しています。WSGI準拠のミドルウエアも今後増えていくことでしょう。多くのフレームワークやミドルウエアがWSGIに準拠すれば、Web開発に必要ないろいろな部品を自由に選び、利用できる夢のような環境が実現できます。

WSGIでは、次期仕様といえる「WSGI 2.0」の仕様を策定中です。より自由度が高く、より適用範囲の広い仕様になるようです。

INDEX

■記号

& …… 50
? …… 49
__def()__ …… 37
__init()__ …… 36
__name__ …… 176
__repr__() …… 184
__setattr__() …… 180
__str__() …… 92
< …… 7
<input> …… 62
<option> …… 63
<select> …… 63
<textarea> …… 63
> …… 7
127.0.0.11 …… 42
8ビット文字列 …… 68

■A～B

action …… 56
and …… 29
Archetypes …… 304
Authorization …… 252
BASE64エンコード …… 253
base64モジュール …… 253
BaseHTTPRequestHandler …… 234
BaseHTTPServer …… 234
basestring …… 75
BASIC認証 …… 251

■C

CGI …… 15, 233
cgi-bin …… 45
CGIHTTPServer …… 44, 234
cgitbモジュール …… 97
cgiモジュール …… 50
checkbox …… 64
checked …… 63
CherryPy …… 301
CJK Codecs …… 26
classmethod …… 175
class文 …… 35
close()メソッド …… 119
CMF …… 304
CMS …… 300, 304
connect()関数 …… 118
Content-Type …… 46, 78
Cookie …… 259, 261
Cookieモジュール …… 260
CREATE TABLE …… 112
CRUD …… 212
cursor()メソッド …… 119
datetimeモジュール …… 47

■D～F

DBAPI …… 117
decode()メソッド …… 39
def …… 30
DELETE …… 115
Django …… 296
DTML …… 305
ElementTreeモジュール …… 94
elif …… 29
else …… 29
enumerate()関数 …… 74
eval()関数 …… 157
execure()メソッド …… 119
Exeption …… 206
expires …… 260
fetchall()メソッド …… 119
fetchone()メソッド …… 119
FieldStrage()関数 …… 51
float()関数 …… 67
for文 …… 30

from文 ………… 33

■G～I
Genshi ………… 302
GET ………… 60
getfilst()メソッド ………… 75
getfirst()メソッド ………… 75
getvalue()メソッド ………… 52
GETメソッド ………… 81
GETリクエスト ………… 49
GUI ………… 5
HTML ………… 7
HTTP ………… 10, 76
HTTPServer ………… 234
id ………… 112
if文 ………… 29
import文 ………… 33
in ………… 30
INSERT ………… 113
int()関数 ………… 54, 66
isdigit()メソッド ………… 55, 67
isinstance()関数 ………… 75

■K～N
Kid ………… 302
Last-Modified ………… 92
Linux ………… 21
list ………… 75
LL ………… 22
locals()関数 ………… 156
Mac OS X ………… 21
Matchオブジェクト ………… 204
MD5 ………… 255
method ………… 56
Morselオブジェクト ………… 261
name ………… 56

■O～R
O/Rマッパー ………… 131, 171
or ………… 29
ORDER BY ………… 184
Page Template ………… 305
path ………… 260
Perl ………… 22
PHP ………… 22
pickleモジュール ………… 103
Plone ………… 304
POST ………… 60
POSTメソッド ………… 83
POSTリクエスト ………… 49
print文 ………… 46
pysqlteモジュール ………… 116
radio ………… 63
reset ………… 63
return文 ………… 31
RSS ………… 86
Ruby ………… 22

■S
SELECT ………… 114
selected ………… 60
self ………… 36
Set-Cookie ………… 261
SHA-1 ………… 255
SimpleCookieクラス ………… 260
SimpleHTTPRequestHandler 234
SimpleHTTPServer ………… 41
SQL ………… 110, 113
SQLAlchemy ………… 301
SQLite ………… 110
sqlite3 ………… 116
SQLObject ………… 301
SQLインジェクション ………… 122, 276
startswith()メソッド ………… 204
str()関数 ………… 66
stringモジュール ………… 136

■T～W
Templateクラス ………… 136
text ………… 56
Turbo Gears ………… 300
type ………… 56
UI ………… 5
unicode()関数 ………… 38
UPDATE ………… 114
URL ………… 13
urlencode()関数 ………… 82
URLエンコード ………… 82, 272
Web ………… 2
Webサーバ ………… 13, 40
WHERE ………… 114
while文 ………… 30
Widgets ………… 303
Windows ………… 20
WSGI ………… 306
wsgirefモジュール ………… 307

■X～Z
XML型 ………… 149
yield ………… 184
yum ………… 22
zip()関数 ………… 71
ZODB ………… 305
Zope ………… 295

■あ行
アクセサ ………… 170
アトリビュート ………… 8, 36
アプリケーション ………… 3
アプリケーションサーバ ………… 15
アンパック代入 ………… 32
イテレータ ………… 184
インタラクティブシェル ………… 23
インデックス ………… 26
インデックスアクセス ………… 247
インデント ………… 28
ウィジット ………… 215
エスケープ ………… 82
エンコード ………… 69
応答コード ………… 77
オブジェクト ………… 35
オブジェクトデータベース ………… 305

■か行
カーソル ………… 116
型の変数 ………… 66
カラム ………… 111
環境変数 ………… 80, 237
関数定義 ………… 30
関数デコレータ ………… 247
キー ………… 27
キーワード引数 ………… 31, 177, 182
クエリ ………… 50, 79
組み込み型 ………… 25
クライアント・サーバ型アプリケーション ………… 11
クラスインスタンス ………… 36
クラスオブジェクト ………… 175
クラスメソッド ………… 175
グラフィカル・ユーザーインターフェース ………… 5
繰り返し変数 ………… 30
クロスサイトスクリプティング ………… 273
継承 ………… 35
計量言語 ………… 22
コードブロック ………… 148
コデックス ………… 26, 37
コネクションオブジェクト ………… 116

コントロール …… 61

■さ行
サブミットボタン …… 62, 223
シーケンス型 …… 25
ジェネレータ …… 184
辞書 …… 27
実体参照文字列 …… 273
出力 …… 3
初期化メソッド …… 36
シリアライズ …… 102
シリアル型 …… 113
数値型 …… 25
スキーム …… 13
スクリプト言語 …… 6, 22
スクリプトファイル …… 34
スタンドアロン型アプリケーション …… 11
ステータス行 …… 236
ステータスコード …… 77
スライス …… 27
スロット …… 147
正規表現 …… 204, 209
整数型 …… 25
静的出力 …… 14, 58
セキュリティホール …… 270
セッション …… 100
接続文字列 …… 117
遷移図 …… 194

■た行
ダイジェスト認証 …… 254
タグ …… 7
多重継承 …… 35
タプル型 …… 27
チェックボックス …… 64
抽象クラス …… 173
データ型 …… 112
データベース …… 109
テーブル …… 110
デコレータ …… 175
テンプレート …… 98, 133
テンプレートエンジン …… 131, 133, 150
動的言語 …… 22
動的出力 …… 15, 58
トークナイザー …… 151
ドキュメンテーション文字列 …… 31, 88
ドキュメントルート …… 42
特殊メソッド …… 37
トップレベルのブロック …… 34
トレースバック …… 47

■な行
名前空間 …… 30, 32, 154
日本Pythonユーザ会 …… 20
入力 …… 3
認証 …… 249
認証状態の継続 …… 250

■は行
パス …… 13, 79
ハッシュ …… 254
ハッシュ関数 …… 254
パラメータ …… 49
バリデーション …… 203
バリデータ …… 205
ハンドラ …… 234
ヒアドキュメント …… 98
引数 …… 31
標準出力 …… 77
標準モジュール …… 18
標準ライブラリ …… 32
ビルトイン型 …… 25
フィールドストレージ …… 51
フィールド値 …… 78
フィールド名 …… 78
フォーマット文字列 …… 98
フォーム …… 9, 55
フォーム認証 …… 257
浮動小数点型 …… 25
プレースホルダ …… 119, 277
フレームワーク …… 294
フロー制御 …… 28
ブロック …… 28
プロトコル …… 76
プロンプト …… 24
ヘッダ …… 46
ヘッダ行 …… 236
ヘッダフィールド …… 78
変数 …… 24
ホスト …… 13

■ま行
マークアップ言語 …… 7
マスターテンプレート …… 147
マップ型 …… 27
マルチスレッド …… 108
マルチバイト文字列 …… 37, 68
ミドルウェア …… 308
メソッド …… 36
メッセージダイジェスト …… 254
メニュー …… 64, 221
文字化け …… 69
文字列型 …… 25
文字列テンプレート …… 136
文字列フォーマット …… 47

■や行
ユーザーインターフェース …… 5
ユニコード型 …… 37
ユニコード文字列 …… 68
ユニコード文字列型 …… 26
呼び出し可能オブジェクト …… 31

■ら行
ラジオボタン …… 63, 221
リクエスト …… 12, 79
リクエストハンドリング …… 295
リクエストライン …… 79
リスト …… 26
リセットボタン …… 63, 223
リレーショナルデータベース …… 109
ループバックアドレス …… 42
例外 …… 206
レスポンス …… 12, 77
ログアウト …… 252

■本書サポートページ

http://isbn.sbcr.jp/41812/

本書をお読みになりましたご感想、ご意見を上記URLからお寄せください。また、本書のサンプルソースコードなどもこちらで配布します。

■注意事項

・本書内の内容の実行については、すべて自己責任のもとでおこなってください。内容の実行により発生したいかなる直接、間接的被害について、筆者およびソフトバンク クリエイティブ株式会社、製品メーカー、購入した書店、ショップはその責を負いません。

・本書の内容に関するお問い合わせに関して、編集部への電話によるお問い合わせはご遠慮ください。

・お問い合わせに関しては、封書のみでお受けしております。なお、質問の回答に関しては原則として著者に転送いたしますので、多少のお時間を頂戴、もしくは返答できない場合もありますのであらかじめご了承ください。また、本書を逸脱したご質問に関しては、お答えしかねますのでご了承ください。

みんなのPython(パイソン) Web(ウェブ)アプリ編(へん)

2007年12月5日　初版第1刷発行

著　者……柴田(しばた) 淳(あつし)

発行者……新田光敏

発行所……ソフトバンク クリエイティブ株式会社

〒107-0052 東京都港区赤坂4-13-13

TEL 03-5549-1200（販売）

http://www.sbcr.jp/

印刷・製本　岩岡印刷工業株式会社

カバーデザイン………… 坂川事務所

カバーイラスト………… かわむらふゆみ

本文デザイン・組版……クニメディア株式会社

落丁本、乱丁本は小社販売局にてお取り替えいたします。

定価はカバーに記載されております。

Printed in Japan ISBN 978-4-7973-4181-2