

# Python実習

...

Webアプリケーション (Flask)

# Flask概要

FlaskはPythonのWebアプリフレームワークである。

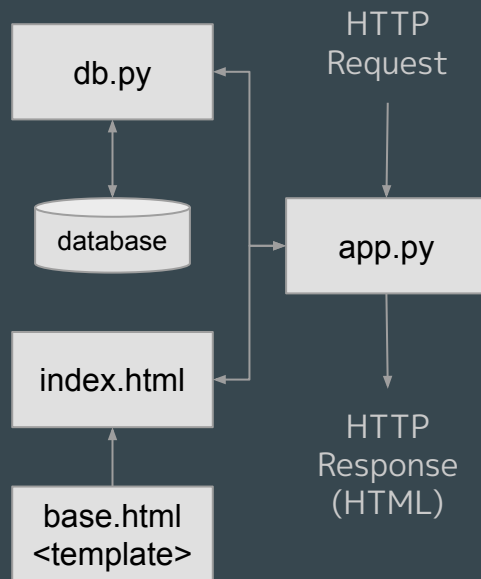
Flaskは、Webアプリケーションの機能が最小限に搭載されているので軽量となる。そのためFlaskを利用するための学習コストは比較的低い。

テンプレートエンジンとしてJinja2を使用している。この機能によりWebアプリケーションの標準的なMVC(Model View Controller)の構造で構築が可能である。

開発用に自身がアプリケーションサーバーとしても起動できるWSGI準拠のツールキットとしてWerkzeug(ヴェルクツォイク)を使用している。あくまで開発用として動作するもので、本番環境ではGunicornやuWSGIなどを利用する。

SQLを使ったORM(Object Relational Mapper)機能は別途ライブラリを利用する必要がある。(SQLAlchemyなど)

【構成例イメージ】



# Webアプリの開発環境

Webアプリ開発の作業用仮想環境を作成する。Pythonで開発を行う際に、一般的には必要なパッケージだけをインストールし利用できるようにするので、目的に応じてプロジェクトを分離させる。

まずはプロジェクトのディレクトリを作成する。(例えば、ここでは「flask」がプロジェクトのディレクトリとなる)

```
[Linux/macOS]  
$ mkdir flask
```

```
[Windows]  
C:\> mkdir flask
```

# Python仮想環境の作成

プロジェクトのディレクトリに移動し、仮想環境を作成し有効化する。(ここでは「web\_env」が仮想環境名となる)

[Linux/macOS]

```
$ cd flask
flask$ python -m venv web_env
flask$ source web_env/bin/activate
(web_env) flask$
```

[Windows]

```
C:\> cd flask
C:\flask> py -m venv web_env
[Windows(cmd)]
C:\flask> web_env\Scripts\activate.bat
[Windows(PowerShell)]
PS C:\flask> web_env\Scripts\Activate.ps1
(web_env) C:\flask>
```

仮想環境が有効になったら、そこにFlaskをインストールする。

[Linux/macOS]

```
(web_env) flask$ python -m pip install flask
```

[Windows]

```
(web_env) C:\flask> py -m pip install flask
```

# Python仮想環境の作成(Windows PowerShellの場合)

Windows PowerShellで仮想環境を有効化するスクリプトの実行がエラーとなる場合は、事前にスクリプトの実行を許可する必要がある。

```
PS C:\> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Force
```

現在のユーザー環境でこのコマンドを1回実行すると2回目以降は不要となる。

今使っているPowerShellウィンドウのみにスクリプトの実行を許可したいときは、先程のコマンドから「CurrentUser -Force」を「Process」に変更する。

```
PS C:\> Set-ExecutionPolicy RemoteSigned -Scope Process
```

# Python仮想環境の作成

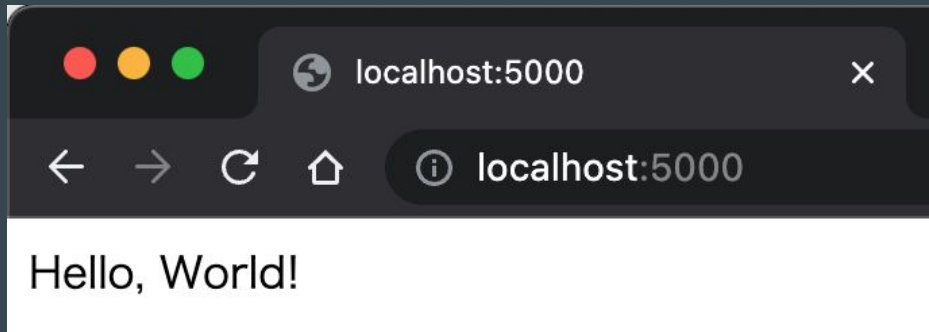
## 【注意点】

外付けドライブに作成した環境を他のPCで有効にする場合は、仮想環境ディレクトリにある「pyvenv.cfg」のhome設定を変更する。使用するPCに応じたPythonのパスに変更する必要がある。

※外付けドライブに作成した仮想環境の再利用は動作不具合が起こりやすいので推奨されない。また異なったOS間での共有はできない。  
仮想環境以外のデータを共有・移行し新たな仮想環境で利用を推奨する。

# 最小のWebアプリ

# 最小のWebアプリ



## ファイル構成

```
flask
└─ hello.py
```

flask/hello.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<p>Hello, World!</p>'
```

flaskディレクトリ配下に左のコードでhello.pyファイルを作成する。Flaskインスタンスでルーティングする。\_\_name\_\_はテンプレートや静的ファイルなどのリソースの場所を特定するために必要となる。route()デコレーターを使って関数をトリガーするURLをFlaskに伝える。この関数はブラウザに表示したい内容を返す。コンテンツタイプはデフォルトでHTMLなので、文字列内のタグはブラウザによってレンダリングされる。



# Webアプリの実行方法

Flaskは開発用として自身がアプリケーションサーバーとして動作することも可能である。あくまで開発用途として利用するものなので、本番環境ではGunicornやuWSGIなどのアプリケーションサーバーを使ってFlaskで記述したWebアプリを動作させる。

この実習では、学習目的のため開発用と同じくFlaskで記述したコード自身を実行しアプリケーションサーバーとしても動作させる。そのためサーバー環境構築は行わない。

# Webアプリの実行方法

ここではFlaskバージョン2.1.x系と2.2.x系の実行方法について説明する。

先程作成したhello.pyを実行するには次のようにコマンドを実行する。

Flask2.1.x

[Linux/macOS]

```
$ export FLASK_APP=hello.py  
$ flask run
```

[Windows(cmd)]

```
> set FLASK_APP=hello.py  
> flask run
```

[Windows(PowerShell)]

```
> $env:FLASK_APP = "hello.py"  
> flask run
```

flaskの実行は「python -m flask」でもよい。

Flask2.2.x

[Linux/macOS/Windows]

```
$ flask --app hello.py run
```

Flask2.2.xは環境変数を設定しない。

コマンド実行後にブラウザに「localhost:5000」とアドレス欄に入力すると画面に「Hello, World!」が表示される。

コマンド実行はflaskディレクトリ配下  
hello.pyと同じ場所で実行する。

# Webアプリの実行方法

デバッグモードを有効にしていると、アプリ動作中にコードを書き換えると、自動的にサーバーをリロードしてくれる。有効化は次のようにする。

Flask2.1.x

[Linux/macOS]

```
$ export FLASK_APP=hello.py  
$ export FLASK_ENV=development  
$ flask run
```

[Windows(cmd)]

```
> set FLASK_APP=hello.py  
> set FLASK_ENV=development  
> flask run
```

[Windows(PowerShell)]

```
> $env:FLASK_APP="hello.py"  
> $env:FLASK_ENV="development"  
> flask run
```

Flask2.2.x

[Linux/macOS/Windows]

```
$ flask --app hello.py run --debug
```

# Webアプリの実行方法

ポート番号を変更することもできる。デフォルトではポート番号5000を使う。ここではポート番号8000を設定する。

Flask2.1.x

[Linux/macOS]

```
$ export FLASK_APP=hello.py  
$ export FLASK_ENV=development  
$ flask run --port=8000
```

[Windows(cmd)]

```
> set FLASK_APP=hello.py  
> set FLASK_ENV=development  
> flask run --port=8000
```

[Windows(PowerShell)]

```
> $env:FLASK_APP="hello.py"  
> $env:FLASK_ENV="development"  
> flask run --port=8000
```

Flask2.2.x

[Linux/macOS/Windows]

```
$ flask --app hello.py run --debug --port=8000
```

コマンド実行後にブラウザに  
「localhost:8000」とアドレス欄に入力しアクセスする。

# Webアプリの実行方法

アプリを動作させている自身のPCとなるlocalhostだけではなく、ネットワーク上の他のPCからもアクセスできるようにするには、次のようにする。

Flask2.1.x

[Linux/macOS]

```
$ export FLASK_APP=hello.py  
$ export FLASK_ENV=development  
$ flask run --host=0.0.0.0
```

[Windows(cmd)]

```
> set FLASK_APP=hello.py  
> set FLASK_ENV=development  
> flask run --host=0.0.0.0
```

[Windows(PowerShell)]

```
> $env:FLASK_APP="hello.py"  
> $env:FLASK_ENV="development"  
> flask run --host=0.0.0.0
```

Flask2.2.x

[Linux/macOS/Windows]

```
$ flask --app hello.py run --debug --host=0.0.0.0
```

# Webアプリの実行方法

Flaskクラスのrun()メソッドを使って実行することもできる。注意点として、本番環境で動作するときにはrun()メソッドは実行しないようにする必要がある。そのため、コード記入時に「if \_\_name\_\_ == '\_\_main\_\_':」の条件下でrun()メソッドを実行するようにする。

Flask2.1.xと2.2.x共通

--省略--

```
if __name__ == '__main__':  
    app.run(host='0.0.0.0', debug=True)
```

このコードをpythonコマンドで実行する。

(例) \$ python hello.py

このコード実行は次のような動作となる。

IPアドレス：localhostおよび公開アドレスでアクセス可能

ポート番号：5000

デバッグモード：有効

# 最小のWebアプリをrun()メソッドで実行

先程作成したhello.pyを修正し、Flaskクラスのrun()メソッドで実行できるようにする。

flask/hello.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<p>Hello, World!</p>'

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

Flask2.1.xと2.2.xのどちらも同じ動作方法となる。

修正したhello.pyをpythonコマンドで実行する。  
\$ python hello.py

このコード実行は次のような動作となる。  
IPアドレス：localhostおよび公開アドレスでアクセス可能  
ポート番号：5000  
デバッグモード：有効

以降の実習内容はこの方法でWebアプリを実行する。

ルーティング



# ルーティング

URLパスに応じて特定の処理を行いページを表示するには、route()デコレーターを使ってそのURLに関数をバインドする。

flask/hello2.py

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'トップページ'

@app.route("/hello")
def hello():
    return 'Hello, World'

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

## ファイル構成

```
flask
├── hello2.py
├── hello3.py
├── hello4.py
└── hello5.py
```

← → ↺ 🏠 ⓘ localhost:5000

トップページ

← → ↺ 🏠 ⓘ localhost:5000/hello

Hello, World

# URLパスの記述による挙動の違い

URLパスの末尾にスラッシュ「/」を記述するときと、しないときで挙動の違いを操作する。

flask/hello3.py

```
from flask import Flask
app = Flask(__name__)

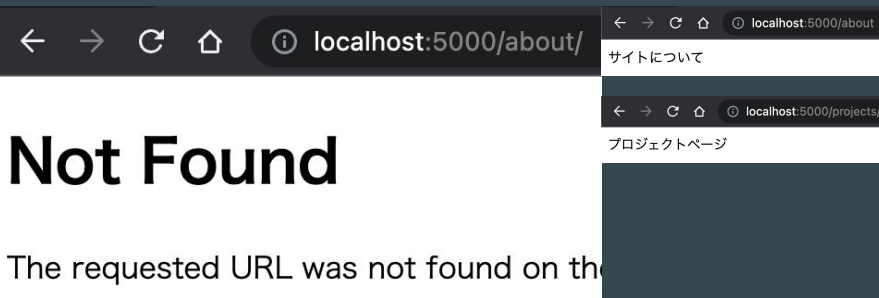
@app.route('/projects/')
def projects():
    return 'プロジェクトページ'

@app.route("/about")
def about():
    return 'サイトについて'

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

スラッシュ(/)を付けずに「/projects」にアクセスすると、Flaskは「/projects/」にリダイレクトする。つまり「/projects」と「/projects/」は同じ関数にバインドされる。

「/about」は「/about/」でアクセスすると、404エラーとなる。



# URLパスを関数の引数にする

記述したURLパスを関数のキーワード引数に受け入れることができる。また、引数に受け入れたデータの型を指定することもできる。

```
from flask import Flask
from markupsafe import escape
app = Flask(__name__)

@app.route('/user/<username>')
def show_user(username):
    return f'ユーザー名 {escape(username)}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    return f'投稿番号 {post_id}'

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    return f'サブパス {escape(subpath)}'

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

flask/hello4.py

<username>のように入力されたパスをその名前の引数で受け入れることができる。入力されたパスを扱うときは、HTMLエスケープして攻撃から防御する必要がある。また、入力されたパスのデータ型を指定することもできる。次の表はデータ型一覧である。

string	スラッシュ(/)なしの文字列を受け入れる
int	正の整数を受け入れる
float	正の浮動小数点数を受け入れる
path	スラッシュ(/)の付いた文字列を受け入れる
uuid	UUID文字列を受け入れる

# 関数を指定してURLを作成する

url\_for()を使ってバインドしている関数を指定し、そのURLパスを作成する。これを使うことで、コード作成後にデコレーターで指定しているURLを変更してもurl\_for()を使っているところは変更無く動作できるようになる。

flask/hello5.py

```
from flask import Flask, url_for
from markupsafe import escape
app = Flask(__name__)
```

```
@app.route('/')
def index():
    return 'トップページ'
```

```
@app.route('/login')
def login():
    return 'ログイン'
```

app.test\_request\_context()はリクエスト処理中でもPythonシェルを使用しているときでもリクエストを処理しているように動作する。

【出力結果】  
/  
/login  
/login?next=foobar  
/user/Hoge

```
@app.route('/user/<username>')
def profile(username):
    return f'{escape(username)}のプロファイル'
```

```
with app.test_request_context():
    print(url_for('index'))
    print(url_for('login'))
    print(url_for('login', next='foobar'))
    print(url_for('profile', username='Hoge'))
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

# 静的ファイルのURLパス

WebアプリもCSSやJavaScriptなど静的ファイルにアクセスすることが必要になる。静的ファイルへのURLパスを作成するときもurl\_for()を使う。

```
url_for('static', filename='style.css')
```

静的ファイルを置く場所はデフォルトではstaticディレクトリ配下となる。上記コードの場合、「static/style.css」へのURLパスを作成する。つまり、staticディレクトリ配下にstyle.cssファイルを置いておく必要がある。

テンプレートエンジンを使った表示

# テンプレートエンジンを使った表示

Jinja2のテンプレートは様々な種類のテキストファイルを生成できる。Webアプリは主にHTMLとなる。テンプレートをレンダリングするには、`render_template()` メソッドを使用する。テンプレートの名前とテンプレートに渡す変数をキーワード引数として指定する。

flask/hello6.py

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

URLパスが「/hello」や「/hello/<name>」のときに`hello()`関数が呼び出される。もし<name>にデータがあった場合は`name`引数に渡される。そして、テンプレートにも`name`引数でデータを渡す。

例えば、「localhost:5000/hello/hoge」とアクセスした場合、「hoge」が`name`引数に渡される。

# テンプレートエンジンを使った表示

Flaskはテンプレートをtemplatesディレクトリ配下で探す。このディレクトリはアプリがモジュールのときは同じ階層で、パッケージのときはパッケージ内に配置する。

flask/templates/hello.html

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
    <h1>Hello, {{ name }}!</h1>
{% else %}
    <h1>Hello, World!</h1>
{% endif %}
```

モジュール

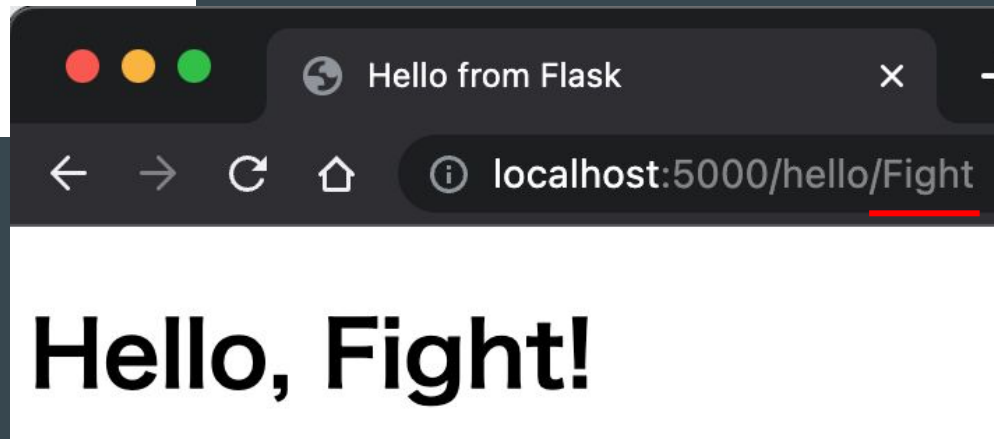
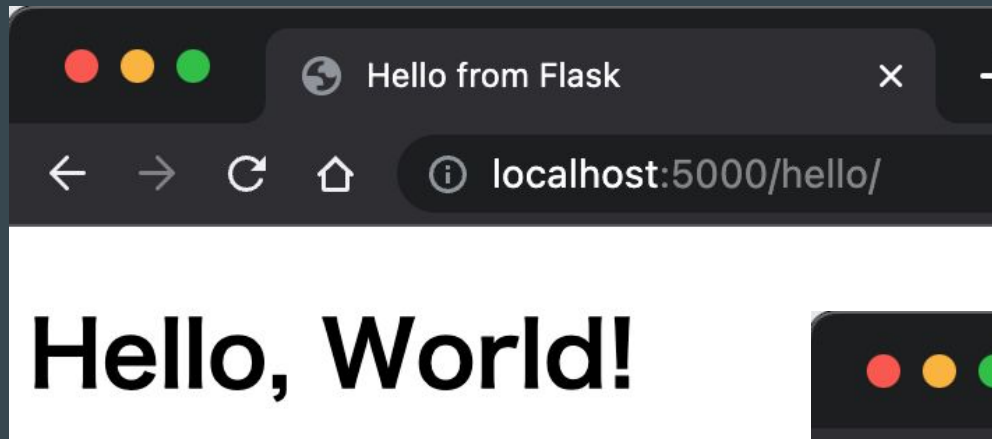
```
flask
├─ hello6.py
└─ templates
    └─ hello.html
```

パッケージ

```
flask
├─ application
│   └─ __init__.py
│       └─ templates
│           └─ hello.html
```



# テンプレートエンジンを使った表示



# テンプレートエンジンを使った表示

テンプレート内で別のテンプレートを読み込むことで、HTMLの共通部分を複数記述せずに一つにすることができる。



## ファイル構成

```
flask
├── app1.py
└── templates
    ├── app1.html
    └── base.html
```

# テンプレートエンジンを使った表示

HTMLテンプレートを作成して、Pythonコード内のデータを表示する。

flaskディレクトリ配下に次のコードでapp1.pyを作成する。

flask/app1.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    menu = [
        {'item': 'コーヒー', 'price': 340},
        {'item': 'パンケーキ', 'price': 750},
        {'item': 'クレープ', 'price': 600},
    ]
    return render_template('app1.html', menu=menu)

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

URLパスがドメイン(web)ルートをリクエストされたとき、app1.htmlをレスポンスで返す。

menu変数にリストを格納し、app1.htmlテンプレートにmenu変数を渡す。

# テンプレートエンジンを使った表示

templatesディレクトリ配下にHTML共通部分のbase.htmlを作成する。

flask/templates/base.html

```
<!DOCTYPE html>
<html lang="ja">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Flask実習</title>
</head>
<body>
  <h1>
    {% block title %}
    <!-- ここにタイトルを表示 -->
    {% endblock %}
  </h1>
  {% block contents %}
  <!-- ここにコンテンツを表示 -->
  {% endblock %}
</body>
</html>
```

見出し1のタイトルを表示するためのブロック

画面に内容を表示するためのブロック

# テンプレートエンジンを使った表示

templatesディレクトリ配下に画面に表示されるapp1.htmlを作成する。

flask/templates/app1.html

```
{# base.htmlを継承する #}  
{% extends "base.html" %}  
  
{# タイトルのブロックを記述 #}  
{% block title %}  
カフェメニュー  
{% endblock %}
```

テーブルにメニューと価格を表示する

```
{# コンテンツのブロックを記述 #}  
{% block contents %}  
  <div>  
    <table border="solid">  
      <thead>  
        <tr>  
          <th>メニュー</th>  
          <th>価格</th>  
        </tr>  
      </thead>  
      <tbody>  
        {% for m in menu %}  
          <tr>  
            <td>{{m.item}}</td>  
            <td>{{m.price}}</td>  
          </tr>  
        {% endfor %}  
      </tbody>  
    </table>  
  </div>  
{% endblock %}
```

GETを使ってデータを操作

# GETを使ってデータを操作

← → ↻ 🏠 ⓘ localhost:5000

## Flask実習 GETサンプル

名前:



## ファイル構成

```
flask
├── app2.py
└── templates
    ├── base.html
    ├── app2.html
    └── app2hello.html
```

← → ↻ 🏠 ⓘ localhost:5000/hello?name=大関

## Flask実習 GETサンプル

大関さん、こんにちは！

# GETを使ってデータを操作

## 【GETとは】

URLのリソースを取り出す（リクエストする）HTTPプロトコルのメソッドである。URLにリクエストパラメータを付加してデータを渡す。

例) `https://foobar.com/sample.html?id=8&age=23`

パラメータは「キー=値」のセットを「?」以降に記述する。複数ある場合は「&」で接続して複数指定する。



# GETを使ってデータを操作

flaskディレクトリ配下に次のコードでapp2.pyを作成する。

flask/app2.py

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('app2.html')

@app.route('/hello')
def hello():
    name = request.args.get('name')
    if name is None or name == '':
        name = '名無し'
    return render_template('app2hello.html', name=name)

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

ドメイン(web)ルートではapp2.htmlを表示

helloパスにアクセスしたときに、GET送信されたデータのnameパラメーターを取得しapp2\_hello.htmlに渡す  
nameパラメーターにデータがなかった場合はname変数は「名無し」となる

# GETを使ってデータを操作

templatesディレクトリ配下に次のコードで  
app2.htmlを作成する。

```
flask/templates/app2.html
```

入力フィールドと送信ボタンを表示

```
{# base.htmlを継承する #}  
{% extends "base.html" %}  
  
{# タイトルのブロックを記述 #}  
{% block title %}  
Flask実習 GETサンプル  
{% endblock %}  
  
{# コンテンツのブロックを記述 #}  
{% block contents %}  
<form action="/hello" method="GET">  
  名前 : <input type="text" name="name">  
  <button type="submit">送信</button>  
</form>  
{% endblock %}
```

# GETを使ってデータを操作

templatesディレクトリ配下に次のコードで  
app2hello.htmlを作成する。

flask/templates/app2hello.html

入力した名前を表示する

```
{# base.htmlを継承する #}  
{% extends "base.html" %}
```

```
{# タイトルのブロックを記述 #}  
{% block title %}  
Flask実習 GETサンプル  
{% endblock %}
```

```
{# コンテンツのブロックを記述 #}  
{% block contents %}  
<p style="font-size: 32px;">{{name}}さん、  
こんにちは！</p>  
{% endblock %}
```

# GETを使ってデータを操作

テンプレートはデフォルトでテキストをHTML  
エスケープする。



← → ↻ 🏠 ⓘ localhost:5000

## Flask実習 GETサンプル

名前:

送信



← → ↻ 🏠 ⓘ localhost:5000/hello?name=<script>alert('XSS');</script>

## Flask実習 GETサンプル

<script>alert('XSS');</script>さん、こんにちは！

POSTを使ってデータを操作

# POSTを使ってデータを操作

## ファイル構成

```
flask
├── app3.py
├── message.txt
├── templates
│   ├── app3.html
│   └── base.html
```

← → ↻ 🏠 localhost:5000

## Flask実習 POSTサンプル

### メッセージ

書き込みはありません。

### メッセージ内容を更新

書込

← → ↻ 🏠 localhost:5000

## Flask実習 POSTサンプル

### メッセージ

書き込みはありません。

### メッセージ内容を更新

Flask(フラスコ)は、プログラミング言語Python用の軽量なWebアプリケーションフレームワークである。標準で提供する機能を最小限に保っているため、自身を「マイクロフレームワーク」と読んでいる。

書込

← → ↻ 🏠 localhost:5000

## Flask実習 POSTサンプル

### メッセージ

Flask(フラスコ)は、プログラミング言語Python用の軽量なWebアプリケーションフレームワークである。標準で提供する機能を最小限に保っているため、自身を「マイクロフレームワーク」と読んでいる。

### メッセージ内容を更新

書込

# POSTを使ってデータを操作

## 【POSTとは】

クライアントがサーバーへデータを送信するHTTPプロトコルのメソッドである。HTTPヘッダにデータを付加して送信する。

GETの場合はURLにデータを付加するため目視可能だが、POST送信はデータをHTTPヘッダのあとに送信するため、フォームのデータを秘匿する目的や容量の多いデータ、ファイルデータを送信するときに利用される。

# POSTを使ってデータを操作

flaskディレクトリ配下に次のコードでapp3.pyを作成する

flask/app3.py

```
from flask import Flask, render_template, request, redirect
import os

app = Flask(__name__)

MESSAGE = './message.txt'
```

トップにアクセスした際はファイルに保存されているメッセージを表示する（なければデフォルトを表示）  
書込ボタンを押すと入力エリアに書かれた文字列をファイルに保存しトップにリダイレクトする

```
@app.route('/')
def index():
    msg = '書き込みはありません。'
    if os.path.exists(MESSAGE):
        with open(MESSAGE, 'r') as f:
            msg = f.read()
        msg = msg.replace('\n', '<br>')

    return render_template('app3.html', msg=msg)

@app.route('/write', methods=['POST'])
def write():
    if 'msg' in request.form:
        msg = str(request.form['msg'])
        with open(MESSAGE, 'w') as f:
            f.write(msg)
    return redirect('/')

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```



# POSTを使ってデータを操作

templatesディレクトリ配下に次のコードでapp3.htmlを作成する。

```
flask/templates/app3.html
```

```
{# base.htmlを継承する #}  
{% extends "base.html" %}
```

```
{# タイトルのブロックを記述 #}  
{% block title %}  
Flask実習 POSTサンプル  
{% endblock %}
```

```
{# コンテンツのブロックを記述 #}  
{% block contents %}  
<h2>メッセージ</h2>  
<div style="background:yellow;  
padding:1rem; max-width:  
700px;">{{msg}}</div>  
<h2>メッセージ内容を更新</h2>  
<form action="/write" method="POST">  
  <textarea name="msg" rows="5"  
cols="50"></textarea><br>  
  <button type="submit">書込</button>  
</form>  
{% endblock %}
```

メッセージを表示するブロック（背景色黄色、内側余白1文字、最大横幅700px）と、入力エリア（5行50桁）と書込ボタンを表示

# POSTを使ってデータを操作

テンプレートはデフォルトでテキストをHTML  
エスケープする。

← → ↺ ⌂ ⓘ localhost:5000

## Flask実習 POSTサンプル

### メッセージ

書き込みはありません。

### メッセージ内容を更新

書込



← → ↺ ⌂ ⓘ localhost:5000

## Flask実習 POSTサンプル

### メッセージ

<script>alert('XSS');</script>

### メッセージ内容を更新

書込

HTMLエスケープせずに、タグを機能させたい場合は、テンプレートに渡した変数に次のように「safe」を設定するとよい。

--省略--

```
<div style="background:yellow;
padding:1rem; max-width:
700px;">{{msg|safe}}</div>
<h2>メッセージ内容を更新</h2>
```

--省略--

掲示板サービス

# 掲示板サービス

← → ↺ 🏠 ⓘ localhost:5000/login

## Flask実習 掲示板

### ログイン

ユーザー名:

パスワード:



Flask実習

localhost:5000

## Flask実習 掲示板

ログイン中 - ほげ

[ログアウト](#)

## Flask実習 掲示板

ログイン中 - ほげ

[ログアウト](#)

今日の飲み会は18:00に集合です。よろしくお願いします。



Flask実習

localhost:5000

## Flask実習 掲示板

ログイン中 - ほげ

[ログアウト](#)

ほげ - 2023/03/24 11:18

今日の飲み会は18:00に集合です。よろしくお願いします。

# 掲示板サービス

掲示板サービスは、ログインし書き込みをするとユーザー名と書き込んだ内容を表示する。

ここではログインするユーザー名はどのような名前でも良く、パスワードは全て「1234」とする。

ログイン中はセッションを確立し、セッション有効期限を設定する。

## 【各種ファイル】

app4.py：メインコードとなりURLアクセスに対する動作を定義する

app4data.py：書き込みをJSONファイルに保存・読み込みする

app4auth.py：ログインとログアウトを処理する

app4top.html：書き込み画面を表示する

app4auth.html：ログイン画面を表示する

msg.html：エラーメッセージを表示する

base.html：すべてのHTMLファイルのテンプレート

bbs.json：投稿内容を保存する

## ファイル構成

```
flask
├── app4.py
├── app4data.py
├── app4auth.py
├── data
│   └── bbs.json
├── templates
│   ├── base.html
│   ├── app4.html
│   ├── app4auth.html
│   └── msg.html
```

※「data」ディレクトリを作成しておく

# 掲示板サービス

flask/app4.py

```
from flask import Flask, render_template
from flask import request, redirect
from datetime import timedelta
import app4auth #ログイン管理
import app4data #データ入出力
```

```
app = Flask(__name__)
app.secret_key = 'Ms44EsJIk6AoVD3g' #セッション情報を暗号化するためのキー
app.permanent_session_lifetime = timedelta(minutes=10) #セッション有効期限 10分
```

```
@app.route('/')
def index():
    if not app4auth.is_login():
        return redirect('/login')
    return render_template('app4.html',
        user=app4auth.get_user(),
        data=app4data.load_data())
```

次のコードでapp4.pyを作成する。

処理に必要なライブラリをインポートする。  
セッション情報を暗号化するためのキーを設定する。  
セッション有効期限を10分にする。

トップにアクセスしたときにログインしていた場合はindex.htmlを表示する。ログインしていない場合はloginパスへリダイレクトする。

# 掲示板サービス

app4.pyの続き

```
@app.route('/login')
def login():
    return render_template('app4auth.html')

@app.route('/check_login', methods=['POST'])
def check_login():
    user, password = None, None
    if 'user' in request.form:
        user = request.form['user']
    if 'password' in request.form:
        password = request.form['password']
    if (user is None) or (password is None):
        return redirect('/login')
    if not app4auth.login(user, password):
        return show_msg('ログインに失敗しました')
    return redirect('/')
```

```
@app.route('/logout')
def logout():
    app4auth.logout()
    return show_msg('ログアウトしました')
```

ログアウトのリンクをクリックしたらlogoutパスでログアウト処理をしmsg.htmlで結果を表示する。

loginパスへリダイレクトされるとapp4auth.htmlを表示する。  
ログイン画面のフォームから送信された内容はPOST送信され、check\_loginパスでチェック処理する。  
ログインに失敗した場合はmsg.htmlでエラーを表示し、成功した場合はトップページへリダイレクトする。

# 掲示板サービス

app4.pyの続き

```
@app.route('/write', methods=['POST'])
def write():
    if not app4auth.is_login():
        return redirect('/login')
    bbs = request.form.get('bbs', '')
    if bbs == '':
        return show_msg('書き込みが空でした。')
    app4data.save_data_append(
        user=app4auth.get_user(),
        text=bbs)
    return redirect('/')

def show_msg(msg):
    return render_template('msg.html', msg=msg)

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

トップページで書き込みが実施されたら、writeパスへリダイレクトされるので、入力エリアから書き込み内容を取得して空でなければdataディレクトリ配下のbbs.jsonファイルへ追記する。追記するとトップへリダイレクトし、書き込んだ内容を表示する。

show\_msg()関数はmsg.htmlで引数で受け入れた文字列を表示する。



# 掲示板サービス

flask/app4auth.py

次のコードでapp4auth.pyを作成する。

logout()関数はセッションからデータを取り除きTrueを返却する。

get\_user()関数はログイン状態ならセッションからログインしているユーザー名を返却する。

is\_login()関数はログイン状態を返却する。

login()関数はパスワードをチェックし、ログインできればセッションにユーザー名を保存しTrueを返却する。

```
from flask import session

def is_login():
    return 'login' in session

def login(user, password):
    if password != '1234':
        return False
    session['login'] = user
    return True

def logout():
    session.pop('login', None)
    return True

def get_user():
    if is_login():
        return session['login']
    return 'not login'
```

# 掲示板サービス

次のコードでapp4data.pyを作成する。

load\_data()関数はbbs.jsonからデータを読み込む。  
save\_data()関数はbbs.jsonにデータを保存する。  
save\_data\_append()関数はsave\_data()を呼び出し、書き込みを追記する。

```
import os, json, datetime
```

```
flask/app4data.py
```

```
BASE_DIR = os.path.dirname(__file__)
```

```
SAVE_FILE = BASE_DIR + '/data/bbs.json'
```

```
def load_data():
```

```
    if not os.path.exists(SAVE_FILE):
```

```
        return []
```

```
    with open(SAVE_FILE, 'r') as fin:
```

```
        return json.load(fin)
```

```
def save_data(data_list):
```

```
    with open(SAVE_FILE, 'w') as fout:
```

```
        json.dump(data_list, fout, ensure_ascii=False, indent=2)
```

```
def save_data_append(user, text):
```

```
    now = datetime.datetime.now()
```

```
    strtime = f'{now:%Y/%m/%d %H:%M}'
```

```
    data = {'name': user, 'text': text, 'date': strtime}
```

```
    data_list = load_data()
```

```
    data_list.insert(0, data) # 最新の投稿を先頭に挿入
```

```
    save_data(data_list)
```

# 掲示板サービス

templatesディレクトリ配下に次のコードでapp4.htmlを作成する。

flask/templates/app4.html

```
{# base.htmlを継承する #}
{% extends "base.html" %}

{# タイトルのブロックを記述 #}
{% block title %}
Flask実習 掲示板
{% endblock %}
```

app4.htmlで書き込みを行い、保存されているデータがあれば表示する。

```
{# コンテンツのブロックを記述 #}
{% block contents %}
<h2>ログイン中 - {{user}}</h2>
<p><a href="/logout">ログアウト</a></p>
<form action="/write" method="POST">
  <textarea name="bbs" rows="4"
cols="60"></textarea><br>
  <button type="submit">書込</button>
</form>

{% for i in data %}
  <div>
    <p>{{i.name}} - {{i.date}}</p>
    <p>
      {% for s in i.text.split('\n') %}
        {{s}}<br>
      {% endfor %}
    </p>
  </div>
{% endfor %}
{% endblock %}
```

# 掲示板サービス

```
{# base.htmlを継承する #}  
{% extends "base.html" %}
```

```
{# タイトルのブロックを記述 #}  
{% block title %}  
Flask実習 掲示板  
{% endblock %}
```

```
{# コンテンツのブロックを記述 #}  
{% block contents %}  
<h2>ログイン</h2>  
<form action="/check_login" method="POST">  
  ユーザー名 : <input type="text" name="user"><br>  
  パスワード : <input type="password" name="password"><br>  
  <button type="submit">送信</button>  
</form>  
{% endblock %}
```

templatesディレクトリ配下に次のコードで  
app4auth.htmlを作成する。

```
flask/templates/app4auth.html
```

app4auth.htmlでログイン画面を表示する。

# 掲示板サービス

templatesディレクトリ配下に次のコードでmsg.htmlを作成する。

flask/templates/msg.html

```
{# base.htmlを継承する #}
{% extends "base.html" %}

{# タイトルのブロックを記述 #}
{% block title %}
Flask実習 掲示板
{% endblock %}

{# コンテンツのブロックを記述 #}
{% block contents %}
<h2>{{msg}}</h2>
<div><a href="/">トップページへ</a></div>
{% endblock %}
```

msg.htmlでエラーなどを表示する。msg変数に格納された文字列を表示する。

## Flask実習 掲示板

ログアウトしました

[トップページへ](#)

## Flask実習 掲示板

ログインに失敗しました

[トップページへ](#)

# 掲示板サービス

書き込みを保存するJSONファイルのフォーマットは以下となる。

```
[
  {
    "name": "Dyson",
    "text": "OK. Bring the vacuum cleaner we developed to the drinking party.",
    "date": "2023/01/10 22:27"
  },
  {
    "name": "ほげ",
    "text": "今日の飲み会は18:00に集合です。よろしくお願いします。",
    "date": "2023/03/24 11:18"
  }
]
```

"name":ユーザー名、"text":書き込み内容、"date":書き込み日時



# 掲示板サービスの改造

# 掲示板サービスの改造

先程作成した掲示板サービスを次の内容で改造する。

- 新規ユーザーを登録できるようにする（登録ページを追加する）
- 登録するユーザー名とパスワードをJSONファイルに保存する
- パスワード保存の際、ハッシュ化する(ソルトを加える)

追加・修正したコードとユーザー登録用のJSONファイルは新しく作成する。それ以外のファイルはそのまま使用する。

## 【修正・追加するファイル】

app5.py：app4.pyをユーザー登録できるように修正する

app5auth.py：app4auth.pyにユーザー登録処理を追加する

app5auth.html：app4auth.htmlにユーザー登録リンクを追加する

app5signup.html：ユーザー登録ページを表示

users.json：ユーザー名とパスワードを保存する

## ファイル構成

```
flask
├── app5.py
├── app4data.py
├── app5auth.py
├── data
│   ├── bbs.json
│   └── users.json
├── templates
│   ├── base.html
│   ├── app4.html
│   ├── app5auth.html
│   ├── app5signup.html
│   └── msg.html
```



# 掲示板サービスの改造

flask/app5.py

app4.pyを修正しapp5.pyを作成する。

```
--省略--
import app5auth #ログイン管理
--省略--

@app.route('/')
def index():
    if not app5auth.is_login():
        return redirect('/login')
    return render_template('app4.html',
        user=app5auth.get_user(),
        data=app4data.load_data())

@app.route('/login')
def login():
    return render_template('app5auth.html')
```

```
--省略--
@app.route('/check_login', methods=['POST'])
def check_login():
    --省略--
    if not app5auth.login(user, password):
        return show_msg('ログインに失敗しました')
    return redirect('/')

@app.route('/logout')
def logout():
    app5auth.logout()
    return show_msg('ログアウトしました')
```

# 掲示板サービスの改造

```
@app.route('/write', methods=['POST'])
def write():
    if not app5auth.is_login():
        return redirect('/login')
    bbs = request.form.get('bbs', '')
    if bbs == '':
        return show_msg('書込みが空でした。')
    app4data.save_data_append(
        user=app5auth.get_user(),
        text=bbs)
    return redirect('/')
```

/signup/パスでGETとPOSTの両方を受け入れる  
signup()関数を追加する。この関数でユーザー登録  
処理をする。GETのときは登録ページを表示し、  
ユーザー名とパスワードがPOST送信されたときに  
その内容を保存する。

```
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        user, password = None, None
        if 'user' in request.form:
            user = request.form['user']
        if 'password' in request.form:
            password = request.form['password']
        if (user is None) or (password is None):
            return show_msg('登録に失敗しました。')
        if not app5auth.add_user(user, password):
            return show_msg('登録済みユーザーです。')
        app5auth.login(user, password)
        return redirect('/')
    else:
        return render_template('app5signup.html')
```

ユーザー登録できれば、そのままログインする。

# 掲示板サービスの改造

app4auth.pyを修正しapp5auth.pyを作成

```
from flask import session
import os, json, hashlib
```

flask/app5auth.py

```
BASE_DIR = os.path.dirname(__file__)
USER_FILE = BASE_DIR + '/data/users.json'
SALT = 'hMLfe:i32n5j#Aiz'
```

ソルトを加える

```
def password_hash(password):
    code = password + SALT
    code = code.encode('utf-8')
    return hashlib.sha256(code).hexdigest()
```

```
def password_verify(password, hash):
    verify = password_hash(password)
    return (verify == hash)
```

```
def load_users():
    if os.path.exists(USER_FILE):
        with open(USER_FILE, 'r',
            encoding='utf-8') as fin:
            return json.load(fin)
    return {}
```

```
def save_users(users):
    with open(USER_FILE, 'w',
        encoding='utf-8') as fout:
        json.dump(users, fout,
            ensure_ascii=False, indent=2)
```

```
def add_user(user, password):
    users = load_users()
    if user in users:
        return False
    users[user] = password_hash(password)
    save_users(users)
    return True
```

# 掲示板サービスの改造

app4auth.htmlを修正しapp5auth.htmlを作成する。

templates/app5auth.html

```
--省略--
{% block contents %}
<h2>ログイン</h2>
<p><a href="/signup">ユーザー登録</a></p>
<form action="/check_login" method="POST">
  ユーザー名 : <input type="text" name="user"><br>
  パスワード : <input type="password" name="password"><br>
  <button type="submit">送信</button>
</form>
{% endblock %}
--省略--
```

# 掲示板サービスの改造

templateディレクトリ配下にapp5signup.html  
を新たに作成する。

templates/app5signup.html

```
{% extends "base.html" %}
{% block title %}
Flask実習 掲示板
{% endblock %}

{% block contents %}
<h2>ユーザー登録</h2>
<p><a href="/login">ログイン</a></p>
<form action="/signup" method="POST">
  ユーザー名 : <input type="text" name="user"><br>
  パスワード : <input type="password" name="password"><br>
  <button type="submit">登録</button>
</form>
{% endblock %}
```

# 掲示板サービスの改造

## Flask実習 掲示板

### ログイン

[ユーザー登録](#)

ユーザー名 :

パスワード :

送信

## Flask実習 掲示板

### ユーザー登録

[ログイン](#)

ユーザー名 :

パスワード :

登録

data/users.json

```
{
  "hoge": "4ca3bf6dc597d26d569fc6ed9935781a11c7e7cceb9062deb0596870c15915b4",
  "foobar": "37e81ab8022a72efaedeadead79151a4d995364913690ed54b3d3915db65d2ef65e"
}
```

# 掲示板サービスの改造（追記）

パスワードのハッシュ化について、ソルトを加えるとセキュリティが向上するが、製品やサービスを開発する際には、さらに高度なセキュリティ対策のされたハッシュ化を行う。多くの場合において、このような機能を持つ関数などが対象となるプログラム言語の拡張や外部ライブラリなどで提供されている。

FlaskではWerkzeug(ヴェルクツォイク)が次の機能を提供している。

- `generate_password_hash()` #パスワードをハッシュ化する
- `check_password_hash()` #ハッシュ化したパスワードと元のパスワードを比較し、同一であることを判定する

# 掲示板サービスの改造（追記）

## werkzeug.securityの使い方

```
from werkzeug.security import generate_password_hash, check_password_hash
```

```
hash1 = generate_password_hash('1234')  
hash2 = generate_password_hash('1234')  
print(hash1 == hash2) # False
```

同じパスワードをハッシュ化しても毎回違うハッシュデータを生成する

```
print(check_password_hash(hash1, '1234')) # True
```

check\_password\_hash()でチェックすると同一であると判定する。

つまり、ハッシュ化する際のソルト値をランダムにしているのでハッシュ化する度に違うハッシュデータとなる。元パスワードと比較して正しく同一であることをcheck\_password\_hash()で判定できる。

デフォルトでは、hash256を使ってソルト値の長さは16となる。

次のキーワード引数を使って、アルゴリズムとソルト値の長さを指定することができる。

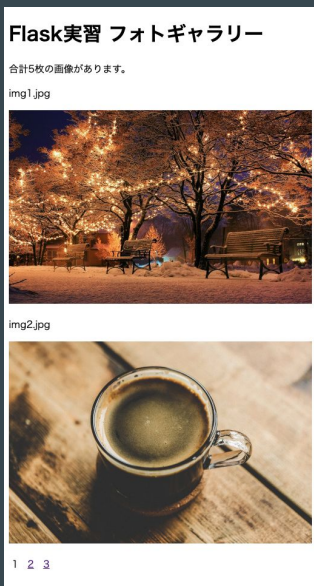
```
hash1 = generate_password_hash('1234', method='sha512', salt_length=10000)  
print(check_password_hash(hash1, '1234')) # True
```



フォトギャラリー

# フォトギャラリー

Webサイトに保存している画像ファイルをブラウザ画面に表示する。1画面に2つの画像を表示する。3つ以上画像が有る場合は、ページリンクを表示してページを遷移して他の画像を表示できるようにする。



ページを表示

# フォトギャラリー

## ファイル構成

```
flask
├── gallery.py
├── static
│   └── images
│       └── ~~.jpg
└── templates
    ├── base.html
    ├── gallery.html
    └── upload.html
```

flaskディレクトリ配下にファイルを構成する

gallery.py：メインプログラム

static：静的なファイルを保存するディレクトリ

images：画像ファイルを保存するディレクトリ

templates：HTMLテンプレートファイルを保存するディレクトリ

base.html：HTMLファイルのベーステンプレート

gallery.html：画像を表示するHTMLページ

upload.html：画像をアップロードするHTMLページ

Flaskでは静的なファイルを扱うときはデフォルトでstaticディレクトリ配下に置くことになっている。画像ファイル以外にもCSSやJavaScriptファイルなどもここに配置する。

これを変更するにはFlaskインスタンス生成時にキーワード引数static\_folderにパスを指定する。実習では、サンプル画像としてimg1~img5.jpgの5つのファイルをimagesディレクトリに格納する。

# フォトギャラリー

URLパラメーター「page」に指定されたページの画像を表示する。

`http://localhost:5000/gallery/?page=1`

1ページに2つの画像を表示するので、imagesディレクトリ配下のファイル数から2つずつのファイル名リストを作成する。リストは昇順にし、画像の合計枚数も画面に表示する。

HTMLはテンプレートを使って作成する。

# フォトギャラリー

flask/gallery.py

```
from flask import Flask, render_template, request
import os
```

```
app = Flask(__name__)
```

```
@app.route('/gallery/')
def gallery():
```

```
    kwargs = {}
```

```
    kwargs['page'] = int(request.args.get('page'))-1
```

```
    kwargs['msg'] = '画像はまだありません。'
```

```
    with os.scandir('./static/images') as it:
```

```
        entries = [entry.name for entry in it if entry.is_file()]
```

```
    entries.sort()
```

gallery.htmlに情報を渡すために  
kwargs辞書を作成する。  
pageキー：「/gallery/?page=1」のアクセスでpageパラメータの値を取得  
msgキー：画像枚数のメッセージ

次ページへつづく

./static/imagesディレクトリ配下にファイルがあればファイル名のリストをentriesに格納する。リストの並びはファイル名の昇順にする。

# フォトギャラリー

つづき

```
cnt = len(entries)
if cnt > 0:
    kwargs['msg'] = f'合計{cnt}枚の画像があります。'
    #2次元リストにページごとに画像ファイル名を格納
    kwargs['entries'] = [entries[i:i+2] for i in range(0,cnt,2)]
    kwargs['pages'] = len(kwargs['entries'])
else:
    kwargs['pages'] = 0

return render_template('gallery.html', **kwargs)

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

imagesディレクトリにファイルが1つ以上あれば以下を処理  
msgキー：合計を表示するメッセージを格納  
entriesキー：ファイル名が2つずつ格納された2次元リストを作成  
pagesキー：総ページ数を格納

render\_template()でgallery.htmlにkwargsに格納した辞書データを渡す。

# フォトギャラリー

HTML共通部分となるテンプレートを作成する。flask配下にtemplatesディレクトリを作成し、そこに次のコードでbase.htmlを作成する。

flask/templates/base.html

```
<!DOCTYPE html>
<html lang="ja">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Flask実習</title>
</head>
<body>
  <h1>
    {% block title %}
    <!-- ここにタイトルを表示 -->
    {% endblock %}
  </h1>
  {% block contents %}
  <!-- ここにコンテンツを表示 -->
  {% endblock %}
</body>
</html>
```

見出し1のタイトルを表示するためのブロック

画面に内容を表示するためのブロック

# フォトギャラリー

画像をブラウザで表示するHTMLを作成する。HTMLの共通部分はbase.htmlを継承する。次のコードでgallery.htmlを作成する。

flask/templates/gallery.html

```
{# base.htmlを継承する #}  
{% extends "base.html" %}  
  
{# タイトルのブロックを記述 #}  
{% block title %}  
Flask実習 フォトギャラリー  
{% endblock %}
```

base.htmlを継承し、見出し1  
のタイトルを記述

```
{# コンテンツのブロックを記述 #}  
{% block contents %}  
<p>{{msg}}</p>  
{% for img in entries[page] %}  
  <p>{{img}}</p>  
    
{% endfor %}  
<p>  
  {% for i in range(1,pages+1) %}  
    {% if i != page+1 %}  
      <span style="margin: 5px;"><a href="/gallery?page={{i}}">{{i}}</a></span>  
    {% else %}  
      <span style="margin: 5px;">{{i}}</span>  
    {% endif %}  
  {% endfor %}  
</p>  
{% endblock %}
```

pageに格納されたインデックスの画像  
ファイルを画面に表示  
画像下に画像のあるページを表示し、  
自身のページ以外はそのページに遷移  
するようにリンクを設定



# フォトギャラリー

./static/imagesディレクトリに画像ファイルを  
保存してブラウザで閲覧できるかを確認する。

プログラム実行>

```
$ python gallery.py
```

```
birdy@StudioBirdy flask % python gallery.py
* Serving Flask app 'gallery'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.99:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 752-363-137
```

URL>

```
http://localhost:5000/gallery/?page=1
```

合計枚数表示

## Flask実習 フォトギャラリー

合計5枚の画像があります。

img1.jpg



img2.jpg



1 2 3

## Flask実習 フォトギャラリー

合計5枚の画像があります。

img3.jpg



img4.jpg

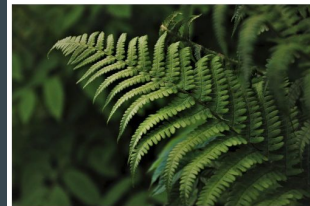


1 2 3

## Flask実習 フォトギャラリー

合計5枚の画像があります。

img5.jpg



1 2 3

ページ表示

# フォトギャラリー

以下の機能を追加する。

- フォトギャラリーにファイルをアップロードする機能を追加
- URLのルート「http://localhost:5000/」にアクセスしたときに、アップロードページを表示



# フォトギャラリー

アップロード機能（フォーム）を表示するページとなるupload.htmlファイルをtemplatesディレクトリ配下に次のコードで作成する。

flask/templates/upload.html

```
{# base.htmlを継承する #}  
{% extends "base.html" %}
```

```
{# タイトルのブロックを記述 #}  
{% block title %}  
Flask実習 フォトギャラリー  
{% endblock %}
```

```
{# コンテンツのブロックを記述 #}  
{% block contents %}  
  {% with messages = get_flashed_messages() %}  
  {% if messages %}  
    <ul style="color: red;">  
      {% for message in messages %}  
        <li>{{ message }}</li>  
      {% endfor %}  
    </ul>  
  {% endif %}  
  {% endwith %}  
  <h2>ファイルアップロード</h2>  
  <form method="post" enctype="multipart/form-data">  
    <input type="file" name="file">  
    <button type="submit">アップロード</button>  
  </form>  
{% endblock %}
```

flash()で表示するエラー内容  
を取得して、赤字でリスト表  
示する

ファイルをアップロードす  
るフォームを表示する

# フォトギャラリー

アップロードページを表示しファイルデータを保存する次のコードをgallery.py  
ファイルに追加する。

flask/gallery.py

```
from flask import Flask, render_template, request, flash, redirect, url_for
import os
```

```
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}
```

← アップロード可能なファイルの拡張子

```
app = Flask(__name__)
```

```
app.secret_key = os.urandom(16)
```

← flash()を利用するためランダムな16文字のsecret\_keyを設定

```
def allowed_file(filename):
```

```
    return '.' in filename and \
```

```
        filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

← ファイルの拡張子をチェックしアップロード可能かを返却する

次ページへつづく

`rsplit(sep=None, maxsplit=-1)`は`sep`を区切り文字とした、文字列中の単語のリストを返す。  
`maxsplit`が与えられた場合、文字列の右端から最大`maxsplit`回分割を行う。

# フォトギャラリー

つづき

```
@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # POST送信にファイルアップロードのデータがあるか確認
        if 'file' not in request.files:
            flash('ファイルデータがありません。 ')
            return redirect(request.url)
        file = request.files['file']
        # ファイルを選択せずアップロードしたときにはファイル名が空となる
        if file.filename == '':
            flash('ファイルが選択されていません。 ')
            return redirect(request.url)
```

flash()でエラーを画面に表示する

次ページへつづく

# フォトギャラリー

つづき

```
# ファイル名をチェックして画像ファイルか確認
if file and allowed_file(file.filename):
    file.save(os.path.join('./static/images', file.filename))
    return redirect(url_for('gallery', page=1))
else:
    flash('画像ファイルではありません。 ')
    return redirect(request.url)

return render_template('upload.html')
```

allowed\_file()で画像ファイルを確認し、ファイルを「./static/images」に保存しgalleryへリダイレクトする。違う場合はflash()でエラーを画面に表示しルートにリダイレクトする。

--省略--

URL入力でのアクセスの場合はupload.htmlを表示する。

url\_for()は特定の関数に対応するURLを構築するのに使用する。キーワード引数を追加することができ、クエリパラメーターとしてURLの後ろに付けられる。この場合gallery関数に関連付けられているURLパスに?page=1のパラメータが付けられる。

# フォトギャラリー

← → ↻ 🏠 ⓘ localhost:5000

## Flask実習 フォトギャラリー

### ファイルアップロード

img6.jpg

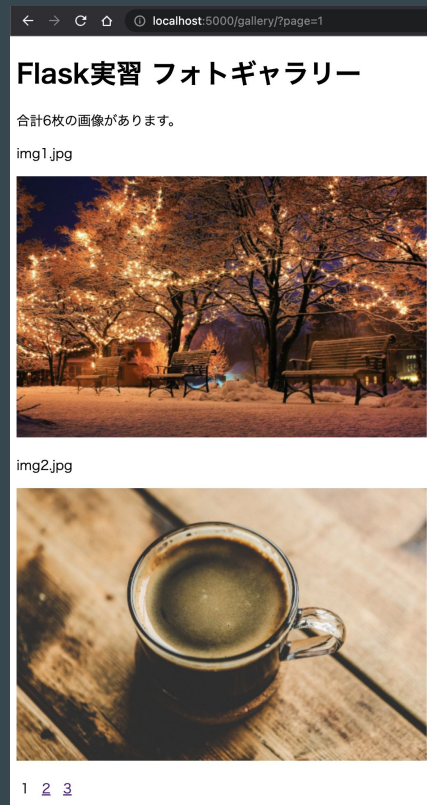
ファイルを選択すると  
ファイル名が表示される

アップロードを押し、正常に  
アップロードできるとギャラ  
リーページにリダイレクトする

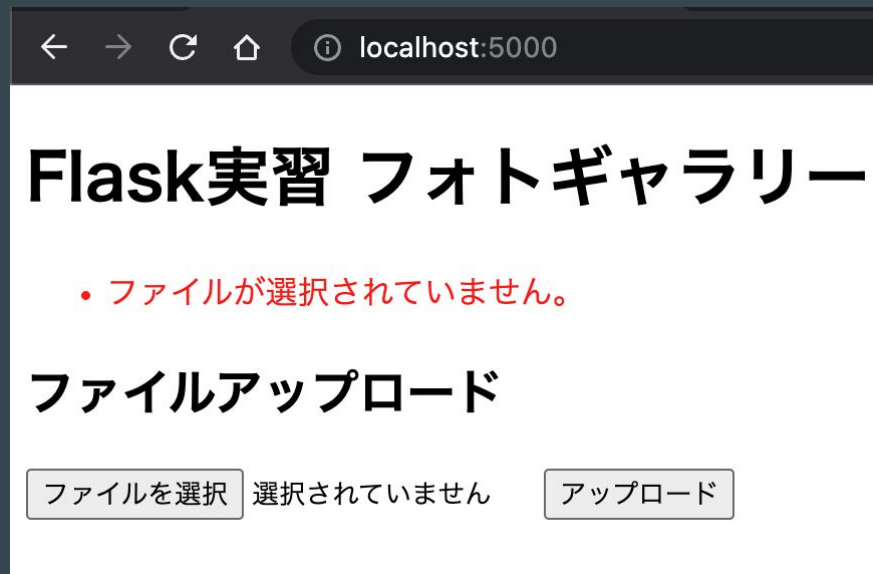
URL)

http://localhost:5000/

アクセスしてアップロー  
ド画面が表示されるか動  
作確認をする



# フォトギャラリー



画像ファイル以外をアップロードしたり、ファイルを選択せずにアップロードするとエラーが表示される。



# フォトギャラリー

より正常に動作させるため、次の修正を行う。

- アップロードページとギャラリーページを行き来できるようにそれぞれのページにリンクを追加する。
- アップロードできるファイル容量を5MBより小さく制限する。
- 同じファイル名がすでにアップロードされていた場合、そのファイル名でアップロードしたときはファイルが上書きされる。そのため、ファイル名に日時を追加しファイルが上書きされないようにする。
- ギャラリーページもALLOWED\_EXTENSIONSに格納した画像ファイルだけを表示するように変更する。

# フォトギャラリー

ギャラリーページへのリンク

flask/templates/upload.html

```
--省略--
{# コンテンツのブロックを記述 #}
{% block contents %}
<p><a href="/gallery/?page=1">ギャラリー
</a></p>
{% with messages = get_flashed_messages()
%}
--省略--
```

アップロードページへのリンク

flask/templates/gallery.html

```
--省略--
{# コンテンツのブロックを記述 #}
{% block contents %}
<p><a href="/">アップロード</a></p>
<p>{{msg}}</p>
--省略--
```

# フォトギャラリー

アップロードするファイルサイズ制限(5MB)とファイル名に日時を追加する。

flask/gallery.py

```
from flask import Flask, render_template, request, flash, redirect, url_for
from datetime import datetime
--省略--
app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 5 * 1024 * 1024
--省略--
@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        --省略--
        if file and allowed_file(file.filename):
            filename = file.filename.rsplit('.', 1)[0]+'_'+datetime.now().strftime('%Y%m%d%H%M%S')
            filename += '.' + file.filename.rsplit('.', 1)[1]
            file.save(os.path.join('./static/images', filename))
            return redirect(url_for('gallery', page=1))
        --省略--
```

MAX\_CONTENT\_LENGTHに制限する  
容量(バイト単位)を設定する。

ファイル名に日時を付け足す。

# フォトギャラリー

imagesディレクトリ配下のファイルはALLOWED\_EXTENSIONSにあるファイル  
だけを表示できるようにする。

flask/gallery.py

```
--省略--
@app.route('/gallery/')
def gallery():
    --省略--
    with os.scandir('./static/images') as it:
        entries = [entry.name for entry in it if entry.is_file() and allowed_file(entry.name)]
    --省略--
```

allowed\_file()関数でTrueが返却されるファイル名の  
画像だけリストに追加する。

# フォトギャラリー

← → ↺ 🏠 ⓘ localhost:5000

## Flask実習 フォトギャラリー

[ギャラリー](#)

### ファイルアップロード

ファイルを選択 選択されていません

アップロード

← → ↺ 🏠 ⓘ localhost:5000/gallery/?page=1

## Flask実習 フォトギャラリー

[アップロード](#)

合計6枚の画像があります。

img1.jpg



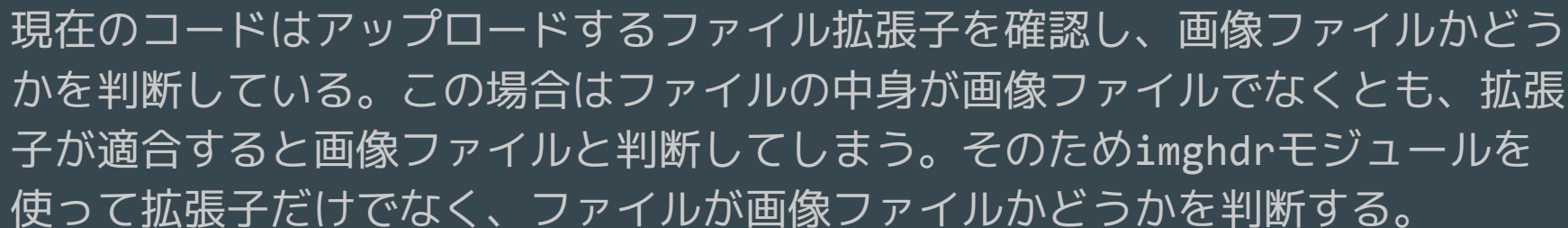
アップロードとギャラリーのリンクがそれぞれのページに表示され、ギャラリーでは画像以外のファイルがimagesディレクトリにあっても画像ファイルだけを画面に表示する。

# フォトギャラリー

セキュリティを向上させ、より利便性を上げるために次の修正を行う。

- 画像ファイルのチェック方法を変更する。
- アップロードした画像ファイルを削除できるようにページを追加する。

# フォトギャラリー

現在のコードはアップロードするファイル拡張子を確認し、画像ファイルかどうかを判断している。この場合はファイルの中身が画像ファイルでなくとも、拡張子が適合すると画像ファイルと判断してしまう。そのためモジュールを使って拡張子だけでなく、ファイルが画像ファイルかどうかを判断する。

```
import imghdr
imghdr.what('sample.png')
>> 'png'
```

imghdrモジュールはファイルやバイトストリームに含まれる画像の形式を決定する。

what()が返す値は画像ファイルの形式となる。  
上の例はPNG形式の値を取得している。  
この実習で扱う形式の種類は右の表となる。  
これら以外の認識できる画像形式は公式サイトを参照。

値	Image format
'gif'	GIF 87a and 89a Files
'jpeg'	JPEG data in JFIF or Exif formats
'png'	Portable Network Graphics

公式サイト (<https://docs.python.org/ja/3/library/imghdr.html?highlight=imghdr#module-imghdr>)

# フォトギャラリー

allowed\_file()関数を画像形式も確認するように修正する。

flask/gallery.py

```
--省略--
import imghdr
--省略--
def allowed_file(file, name=True):
    if name:
        return '.' in file and \
            file.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
    else:
        if '.' in file.filename and \
            file.filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS:
            return imghdr.what(file.stream) in ALLOWED_EXTENSIONS
        else:
            return False
```

キーワード引数nameがTrueのときは引数fileの値はファイル名として処理し、FalseのときはPOST送信でアップロードしたファイルのオブジェクトとして処理する。

imghdr.what()を使いファイルの画像形式を確認する。

ファイル拡張子と画像形式の両方を確認する。



# フォトギャラリー

アップロードするファイルを確認するときの`allowed_file()`の引数を修正する。

flask/gallery.py

```
--省略--
@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        --省略--
        file = request.files['file']
        --省略--
        if file and allowed_file(file, name=False):
            --省略--
```

ここで`allowed_file()`に渡すのはPOST送信で受け取ったファイルのオブジェクトなので、`name`引数を`False`にしてオブジェクトを引数に渡す。

# フォトギャラリー

次にアップロードしたファイルを削除する画面を追加する。

ファイル一覧から削除するファイルをチェックボックスで選択し、削除ボタンを押して削除する。



# フォトギャラリー

削除ページは画像ファイル一覧をチェックボックスで表示する。一覧のファイル情報はギャラリーと同じ方法で取得する。gallery.pyに削除ページのdelete()関数を追加する。

flask/gallery.py

delete()関数はPOST送信で呼ばれる場合と、GET送信で呼ばれる場合の両方を想定する。

--省略--

```
@app.route('/delete', methods=['GET', 'POST'])
def delete():
    kwargs = {}
    kwargs['msg'] = '画像はまだありません。'
    with os.scandir('./static/images') as it:
        entries = [entry.name for entry in it if entry.is_file() and allowed_file(entry.name)]
    entries.sort()
    cnt = len(entries)
    if cnt > 0:
        kwargs['msg'] = f'合計{cnt}枚の画像があります。'
        kwargs['entries'] = entries
```

./static/imagesから画像ファイルを取得し、ファイルが1つ以上ある場合はリストをkwargs['entries']に格納する。

画像ファイルの個数を示すメッセージをkwargs['msg']に格納する。

# フォトギャラリー

POST送信でdelete()関数が呼ばれた場合は、フォームが送信されているのでチェックボックスにチェックされたファイルリストを取得しファイルを削除する。

つづき

--省略--

```
if request.method == 'POST':
    if 'files' in request.form:
        files = request.form.getlist('files')
        for fn in files:
            os.remove('./static/images/'+fn)
        return redirect(request.url)

return render_template('delete.html', **kwargs)
```

POST送信によるチェックされたファイル名のリストを取得し、リストのファイルを削除する。  
そのあと同じページ (/delete) にリダイレクトする。

/deleteのURLが指定されたときは、delete.htmlのテンプレートで表示する。

# フォトギャラリー

削除ページとなるdelete.htmlファイルをtemplatesディレクトリ配下に次のコードで作成する。

flask/templates/delete.html

```
{# base.htmlを継承する #}  
{% extends "base.html" %}
```

```
{# タイトルのブロックを記述 #}  
{% block title %}  
Flask実習 フォトギャラリー  
{% endblock %}
```

ファイル一覧はチェックボックスで表示する。すべてのチェックボックスは関連づけられチェックされた値はリストに格納するので、name属性はすべて同じになる。

```
{# コンテンツのブロックを記述 #}  
{% block contents %}  
<p><a style="margin-right: 20px;" href="/">アップロード</a><a href="/gallery/?page=1">  
ギャラリー</a></p>  
<h2>削除ページ</h2>  
<p>{{msg}}</p>  
<form action="/delete" method="POST">  
    {% for fname in entries %}  
        <div>  
            <input type="checkbox" id="{{fname}}" name="files" value="{{fname}}">  
            <label for="{{fname}}">{{fname}}</label>  
        </div>  
    {% endfor %}  
    <div style="margin-top: 20px;"><button type="submit">削除</button></div>  
</form>  
{% endblock %}
```

# フォトギャラリー

他のページからも削除ページへ移動できるように、リンクを表示する。

flask/templates/upload.html

```
--省略--
{# コンテンツのブロックを記述 #}
{% block contents %}

```

flask/templates/gallery.html

```
--省略--
{# コンテンツのブロックを記述 #}
{% block contents %}

```

# フォトギャラリー

localhost:5000/delete

## Flask実習 フォトギャラリー

[アップロード](#) [ギャラリー](#)

### 削除ページ

合計7枚の画像があります。

- ☐ img1.jpg
- ☐ img2.jpg
- ☐ img3.jpg
- ☐ img4.jpg
- ☒ img5.jpg
- ☐ img6.jpg
- ☒ img7\_20230326174618.jpg

削除



localhost:5000/delete

## Flask実習 フォトギャラリー

[アップロード](#) [ギャラリー](#)

### 削除ページ

合計5枚の画像があります。

- ☐ img1.jpg
- ☐ img2.jpg
- ☐ img3.jpg
- ☐ img4.jpg
- ☐ img6.jpg

削除

選択したファイルが削除されている。

localhost:5000/gallery/?page=3

## Flask実習 フォトギャラリー

[削除ページ](#) [アップロード](#)

合計5枚の画像があります。

img6.jpg



1 2 3