python

# Flask

Flask является микрофреймворком для создания вебсайтов на языке Python

Flask имеет много параметров конфигурации с разумными значениями по умолчанию, и мало предварительных соглашений. По соглашению, шаблоны и статические файлы хранятся в поддиректориях внутри дерева исходных текстов на Python, с названиями *templates* и *static* соответственно. Хотя это можно и поменять, обычно этого делать не стоит, особенно в самом начале работы.

Основная причина почему Flask называется «микрофреймворком» — это идея сохранить ядро простым, но расширяемым

В нем нет абстрактного уровня базы данных, нет валидации форм или всего такого, что уже есть в других библиотеках к которым вы можете обращаться

Однако Flask поддерживает расширения, которые могут добавить необходимую функциональность и имплеминтирует их так, как будто они уже были встроены изначально. В настоящее время уже есть расширения: формы валидации, поддержка закачки файлов, различные технологии аутентификации и многие другие.

# Пример Hello

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

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

if __name__ == "__main__":
    app.run()
```

# Общедоступный сервер

Если вы запустите сервер, вы заметите, что он доступен только с вашего компьютера, а не с любого другого компьютера в сети. Это значение выставлено по умолчанию, потому что в режиме отладки пользователь приложения может выполнить произвольный код на вашем компьютере.

Если вы доверяете пользователям в вашей сети, вы можете сделать сервер общедоступным, просто изменив вызов run(), который должен выглядеть следующим образом:

```
app.run(host='0.0.0.0')
```

# Режим отладки

Метод run() хорош для начала разработки на локальном сервере. Но это потребует ручного перезапуска сервера после каждого изменения в коде

Если включить Debug Mode, сервер будет сам перегружаться после каждого изменения в коде. Еще вы получите полезный отладчик, на тот случай если что-то пойдет не так.

Есть два способа включить режим отладки:

```
app.debug = True
app.run()
```

Или

```
app.run(debug=True)
```

# Маршрутизация

Роутеры строятся с помощью декоратора route()

```
@app.route('/')
def index():
    return 'Index Page'
```

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

# Динамические роутеры

Для добавления переменной части в URL можно пометить эти разделы, как <variable_name>

Дополнительно преобразователь может быть определен путем указания правила <converter:variable_name>

Имеются следующие конверторы

| | |
|---|---|
| int | Принимает целые числа |
| float | то же самое, что int, только с плавающей точкой |
| path | похоже на то, что установлено по умолчанию, но принимает слэши |

```python
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username
```

```python
@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id
```

# Генерация URL

Flask может генерировать URL. Для создания URL, используйте функц `url_for()`.

Она принимает имя функции в качестве первого аргумента, а также ряд ключевых аргументов, каждый из которых соответствует переменной части URL правила

Части неизвестной переменной добавляется к URL в качестве параметров запроса

```
>>> from flask import Flask, url_for
>>> app = Flask(__name__)
>>> @app.route('/')
... def index(): pass
...
>>> @app.route('/login')
... def login(): pass
...
>>> @app.route('/user/<username>')
... def profile(username): pass
...
>>> with app.test_request_context():
...  print url_for('index')
...  print url_for('login')
...  print url_for('login', next='/')
...  print url_for('profile', username='John Doe')
...
/
/login
/login?next=/
/user/John%20Doe
```

Метод test_request_context() говорит Flask, как нужно обрабатывать запрос, даже если мы взаимодействуем через шел Python

# HTTP методы

По умолчанию маршрут реагирует только на ответы GET-запросов, но это можно изменить путем предоставления методов, используя аргументы к декоратору route()

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

Если присутствует GET, тогда HEAD будет добавлен автоматически. Вам не нужно об этом заботиться. Также будьте уверены, что HEAD поддерживает HTTP RFC зависимости, так что вы можете полностью игнорировать HTTP спецификации.

# Статические файлы

Динамическим веб-приложениям также требуются статические файлы. Обычно это css и javascript файлы.

Просто создайте папку с названием static в вашем пакете или рядом с модулем и она будет доступна в /static по применению.

Для генерации адресов для статических файлов, используется специальное имя 'static':

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

Файл будет доступен в файловой системе по пути static/style.css

# Рендеринг шаблонов

Для рендеринга Flask используем Jinja2

Чтобы создать шаблон, можно использовать метод render_template(). Все, что вам нужно сделать, — это указать имя шаблона и переменные, которые вы хотите передать в шаблоны как ключевые аргументы

```python
from flask import render_template

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

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

# WatchDog

Расширение для мониторинга за изменениями файловой системы

Предположим, что нас интересуют изменения по некоему пути /path/to/smth, связанные с созданием, удалением и переименованием файлов и директорий.

Подключаем

```
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
```

Класс Observer выбирается в /observers/__init__.py исходя из возможностей вашей ОС, так что нет необходимости самостоятельно решать, что же выбрать

Класс FileSystemEventHandler является базовым классом обработчика событий изменения

```
class Handler(FileSystemEventHandler):
    def on_created(self, event):
        print event

    def on_deleted(self, event):
        print event

    def on_moved(self, event):
        print event
```

```
observer = Observer()
observer.schedule(Handler(), path='/path/to/smth', recursive=True)
observer.start()
```

Observer является относительно далеким потомком threading.Thread, соотвественно после вызова start() мы получаем фоновый поток, следящий за изменениями

```
# mkdir foo
# touch bar
# mv bar baz
# cd foo/
# mkdir foz
# mv ../baz ./quz
# cp ./quz ../hw
# cd ..
# rm -r ./foo
# rm -f ./*
```

## На выходе

```
<DirCreatedEvent: src_path=/path/to/smth/foo>
<FileCreatedEvent: src_path=/path/to/smth/bar>
<FileMovedEvent: src_path=/path/to/smth/bar, dest_path=/path/to/smth/baz>
<DirCreatedEvent: src_path=/path/to/smth/foo/foz>
<FileMovedEvent: src_path=/path/to/smth/baz, dest_path=/path/to/smth/foo/quz>
<FileCreatedEvent: src_path=/path/to/smth/hw>
<FileDeletedEvent: src_path=/path/to/smth/foo/quz>
<DirDeletedEvent: src_path=/path/to/smth/foo/foz>
<DirDeletedEvent: src_path=/path/to/smth/foo>
<FileDeletedEvent: src_path=/path/to/smth/hw>
```

# Pygame

Набор модулей языка программирования Python, предназначенный для написания компьютерных игр и мультимедиа-приложений. Pygame базируется на мультимедийной библиотеке SDL.

# Что такое SDL?

**Simple DirectMedia Layer (SDL) -** это свободная кроссплатформенная мультимедийная библиотека, реализующая единый программный интерфейс к графической подсистеме, звуковым устройствам и средствам ввода для широкого спектра платформ.

Официально поддерживаются операционные системы: Linux, Microsoft Windows, Mac OS X, iOS и Android.

SDL API доступны для языков: C, C++, C#, VB.NET, D, Ada, Vala, Eiffel, Haskell, Erlang, Euphoria, Java, Lisp, Lua, ML, Pascal, Perl, PHP, Pike, PureBasic, Python и Ruby.

# Пример

```
1. import pygame
2. from pygame.locals import *
3.
4. def init_window():
5.     pygame.init()
6.     window = pygame.display.set_mode((550, 480))
7.     pygame.display.set_caption('My own little world')
8.
9. def main():
10.    init_window()
11.
12. if __name__ == '__main__': main()
```

# Больше примеров

https://github.com/Mekire/pygame-samples

http://pygame.org/tags/example

http://programarcadegames.com/?chapter=example_code

# OpenCV

Библиотека алгоритмов компьютерного зрения, обработки изображений и численных алгоритмов общего назначения с открытым кодом. Реализована на C/C++, также разрабатывается для Python, Java, Ruby, Matlab, Lua и других языков. Может свободно использоваться в академических и коммерческих целях — распространяется в условиях лицензии BSD.

# Применение

Для утверждения общего стандартного интерфейса компьютерного зрения для приложений в этой области. Для способствования росту числа таких приложений и создания новых моделей использования PC.

Сделать платформы Intel  привлекательными для разработчиков таких приложений за счёт дополнительного ускорения OpenCV с помощью Intel® Performance Libraries (Сейчас включают IPP (низкоуровневые библиотеки для обработки сигналов, изображений, а также медиа-кодеки) и MKL (специальная версия LAPACK и FFTPack))

OpenCV способна автоматически обнаруживать присутствие IPP и MKL и использовать их для ускорения обработки

# Поддерживаемые платформы

- Microsoft Windows: компиляторы Microsoft Visual C++ (6.0, .NET 2003), Intel Compiler, Borland C++, Mingw (GCC 3.x).
- Windows RT: портирован на ARM компанией Itseez[3].
- Linux: GCC (2.9x, 3.x), Intel Compiler: «./configure-make-make install», RPM (spec файл включен в поставку).
- Mac OS X: GCC (3.x, 4.x).
- Android.
- iOS.
- Используются C и «облегченный» C++. Прагмы и условная компиляция используются очень ограниченно.

# Основные модули

- opencv_core — основная функциональность. Включает в себя базовые структуры, вычисления (математические функции, генераторы случайных чисел) и линейную алгебру, DFT, DCT, ввод/вывод для XML и YAML и т. д.
- opencv_imgproc — обработка изображений (фильтрация, геометрические преобразования, преобразование цветовых пространств и т. д.).
- opencv_highgui — простой UI, ввод/вывод изображений и видео.
- opencv_ml — модели машинного обучения (SVM, деревья решений, обучение со стимулированием и т. д.).
- opencv_features2d — распознавание и описание плоских примитивов (SURF (русск. / англ.), FAST и другие, включая специализированный фреймворк).
- opencv_video — анализ движения и отслеживание объектов (оптический поток, шаблоны движения, устранение фона).
- opencv_objdetect — обнаружение объектов на изображении (нахождение лиц с помощью алгоритма Виолы-Джонса (*англ.*), распознавание людей HOG и т. д.).
- opencv_calib3d — калибровка камеры, поиск стерео-соответствия и элементы обработки трёхмерных данных.
- opencv_flann — библиотека быстрого поиска ближайших соседей (FLANN 1.5) и обертки OpenCV.
- opencv_contrib — сопутствующий код, ещё не готовый для применения.
- opencv_legacy — устаревший код, сохранённый ради обратной совместимости.
- opencv_gpu — ускорение некоторых функций OpenCV за счет CUDA, создан при поддержке NVidia.

# PIL (Pillow)

Библиотека Python предназначена для работы с растровой графикой.

Разработка библиотеки прекращена (последняя правка датируется 2011 годом).Однако, проект под названием **Pillow**, являющийся форком PIL, развивается и включает, в том числе, поддержку Python 3.x

Этот форк был принят в качестве замены оригинальной библиотеки и включён в некоторые дистрибутивы Linux, включая Debian и Ubuntu (с 13.04)

**Возможности:**

поддержка бинарных, полутоновых, индексированных, полноцветных и CMYK изображений;

поддержка форматов BMP, EPS, GIF, JPEG, PDF, PNG, PNM, TIFF и некоторых других на чтение и запись;

поддержка множества форматов (ICO, MPEG, PCX, PSD, WMF и др.) только для чтения;

конвертирование изображений из одного формата в другой;

редактирование изображений (использование различных фильтров, масштабирование, рисование, матричные операции и т. д.);

# Создание изображения

```
import Image, ImageDraw
text = "Python Imaging Library in Habr :)"
color = (0, 0, 120)
img = Image.new('RGB', (100, 50), color)
imgDrawer = ImageDraw.Draw(img)
imgDrawer.text((10, 20), text)
img.save("pil-example.png")
```

# Вывод формата типа и размера изображения

```
import Image, ImageDraw
img = Image.open('test.png') #открываем картинку
size = img.size #размер картинки
format = img.format #формат картинки
mode = img.mode #мод(RGBA...)
arr = [] #создаем пустой массив
arr.append(size)  #добавляем размер в массив
arr.append(format) #добавляем формат в массив
arr.append(mode)  #добавляем мод в массив
print arr #выводим массив
```

# Конвертация изображения

```
import Image, ImageDraw
img = Image.open('test.png')
img.save('test.gif')
```

# Peewee

Лёгкая, гибкая и очень быстрая ORM на Python

Особенности:

Маленькая,красивый ORM

Написана на Python,с поддержкой Python 2.6+, Python 3.2+

Поддерживает Sqlite,Mysql,PostgreSql

Тонна расширений,доступных в Playhouse

http://docs.peewee-orm.com/en/latest/peewee/playhouse.html

# Пример

```
from peewee import *
import datetime

db = SqliteDatabase('my_database.db', threadlocals=True)

class BaseModel(Model):
    class Meta:
        database = db

class User(BaseModel):
    username = CharField(unique=True)

class Tweet(BaseModel):
    user = ForeignKeyField(User, related_name='tweets')
    message = TextField()
    created_date = DateTimeField(default=datetime.datetime.now)
    is_published = BooleanField(default=True)
```

# Коннект к базе

```
db.connect()
db.create_tables([User, Tweet])
```

# Добавляем запись

```
charlie = User.create(username='charlie')
huey = User(username='huey')
huey.save()

# No need to set `is_published` or `created_date` since they
# will just use the default values we specified.
Tweet.create(user=charlie, message='My first tweet')
```

# Типы полей

| Field Type | Sqlite | Postgresql | MySQL |
|---|---|---|---|
| `CharField` | varchar | varchar | varchar |
| `FixedCharField` | char | char | char |
| `TextField` | text | text | longtext |
| `DateTimeField` | datetime | timestamp | datetime |
| `IntegerField` | integer | integer | integer |
| `BooleanField` | smallint | boolean | bool |
| `FloatField` | real | real | real |
| `DoubleField` | real | double precision | double precision |
| `BigIntegerField` | integer | bigint | bigint |
| `DecimalField` | decimal | numeric | numeric |
| `PrimaryKeyField` | integer | serial | integer |
| `ForeignKeyField` | integer | integer | integer |
| `DateField` | date | date | date |
| `TimeField` | time | time | time |
| `BlobField` | blob | bytea | blob |
| `UUIDField` | not supported | uuid | not supported |

# Requests

Requests — библиотека Python, которая элегантно и просто выполняет HTTP-запросы. Теперь не нужно осваивать urllib2 с излишне сложными программными интерфейсами.

# HTTP-запрос с авторизацией

```
>>> r = requests.get('https://api.github.com', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json'
```

# На Urllib тоже самое

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

import urllib2

gh_url = 'https://api.github.com'
gh_user= 'user'
gh_pass = 'pass'

req = urllib2.Request(gh_url)

password_manager = urllib2.HTTPPasswordMgrWithDefaultRealm()
password_manager.add_password(None, gh_url, gh_user, gh_pass)

auth_manager = urllib2.HTTPBasicAuthHandler(password_manager)
opener = urllib2.build_opener(auth_manager)

urllib2.install_opener(opener)

handler = urllib2.urlopen(req)

print handler.getcode()
print handler.headers.getheader('content-type')

# ------
# 200
# 'application/json'
```

Библиотека requests позволяет отправлять HTTP-запросы HEAD, GET, POST, PUT, PATCH и DELETE

# SH

Является полноправной заменой подпроцессов для Python 2.6 - 3.4, что позволяет вызывать любую программу так, как будто это была функция

https://github.com/amoffat/sh

# Пример

```
from sh import ifconfig
print(ifconfig("wlan0"))
```

```
# checkout master branch
git.checkout("master")

# print(the contents of this directory
print(ls("-l"))

# get the longest line of this file
longest_line = wc(__file__, "-L")
```