

## Установка и настройка Django

### Создайте виртуальную среду для приложения:

Виртуальная среда или **venv** не является неотъемлемой частью разработки на Django. Однако ее рекомендуется использовать, так как она позволяет создать множество виртуальных сред Python на одной операционной системе. Благодаря виртуальной среде приложение может запускаться независимо от других приложений на Python.

В принципе можно запускать приложения на Django и без виртуальной среды. В этом случае все пакеты Django устанавливаются глобально. Однако что если после создания первого приложения выйдет новая версия Django? Если мы захотим использовать для второго проекта новую версию Django, то из-за глобальной установки пакетов придется обновлять первый проект, который использует старую версию. Это потребует некоторой дополнительной работы по обновлению, так как не всегда соблюдается обратная совместимость между пакетами. Если мы решим использовать для второго проекта старую версию, то мы лишимся потенциальных преимуществ новой версии. И использование виртуальной среды как раз позволяет разграничить пакеты для каждого проекта.

Для работы с виртуальной средой в python применяется встроенный модуль **venv**

```
py -m venv venv
```

Для использования виртуальную среду **надо активировать**. И каждый раз, когда мы будем работать с проектом Django, связанную с ним виртуальную среду надо активировать. Например, активируем выше созданную среду, которая располагается в текущем каталоге в папке **venv**. Процесс активации немного отличается в зависимости от операционной системы и от того, какие инструменты применяются. Так, в Windows можно использовать командную строку и PowerShell, но между ними есть отличия.

```
venv\scripts\activate
```

```
• PS C:\Users\Vital\Python\Test2> py -m venv venv  
• PS C:\Users\Vital\Python\Test2> venv\scripts\activate  
◦ (venv) PS C:\Users\Vital\Python\Test2>
```

## Установка Django

После активации виртуальной среды для установки Django выполним в консоли следующую команду

```
python -m pip install Django
```

```
• (venv) PS C:\Users\Vital\Python\Test2> python -m pip install Django
Collecting Django
  Using cached Django-5.1.3-py3-none-any.whl.metadata (4.2 kB)
Collecting asgiref<4,>=3.8.1 (from Django)
  Using cached asgiref-3.8.1-py3-none-any.whl.metadata (9.3 kB)
Collecting sqlparse>=0.3.1 (from Django)
  Using cached sqlparse-0.5.1-py3-none-any.whl.metadata (3.9 kB)
Collecting tzdata (from Django)
  Using cached tzdata-2024.2-py2.py3-none-any.whl.metadata (1.4 kB)
Using cached Django-5.1.3-py3-none-any.whl (8.3 MB)
Using cached asgiref-3.8.1-py3-none-any.whl (23 kB)
Using cached sqlparse-0.5.1-py3-none-any.whl (44 kB)
Using cached tzdata-2024.2-py2.py3-none-any.whl (346 kB)
Installing collected packages: tzdata, sqlparse, asgiref, Django
Successfully installed Django-5.1.3 asgiref-3.8.1 sqlparse-0.5.1 tzdata-2024.2

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
◦ (venv) PS C:\Users\Vital\Python\Test2> █
```

## Создание проекта

При установке Django в папке виртуальной среды устанавливается утилита **django-admin**. А на Windows также исполняемый файл **django-admin.exe**. Их можно найти в папке виртуальной среды, в которую производилась установка Django: на Windows - в подкаталоге **Scripts**

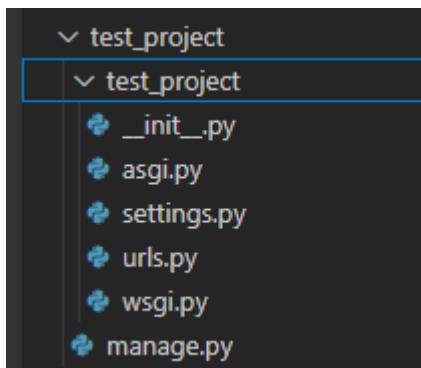
**django-admin** предоставляет ряд команд для управления проектом Django. В частности, для создания проекта применяется команда **startproject**. Этой команде в качестве аргумента передается название проекта.

Итак, создадим первый на Django. Пусть он будет располагаться в той же папке, где располагается каталог виртуальной среды. И для этого вначале активируем ранее созданную виртуальную среду (например, среду `.venv`, которая была создана в прошлой теме, если она ранее не была активирована).

И после активации виртуальной среды выполним следующую команду

```
django-admin startproject test_project
```

```
• (venv) PS C:\Users\Vital\Python\Test2> django-admin startproject test_project
```



Созданный каталог будет состоять из следующих элементов:

- `manage.py`: выполняет различные команды проекта, например, создает и запускает приложение
- `test_project` - собственно папка проекта `test_project`, которая содержит следующие файлы:
  - `__init__.py`: данный файл указывает, что папка, в которой он находится, будет рассматриваться как модуль. Это стандартный файл для программы на языке Python.
  - `settings.py`: содержит настройки конфигурации проекта
  - `urls.py`: содержит шаблоны URL-адресов, по сути определяет систему маршрутизации проекта
  - `wsgi.py`: содержит свойства конфигурации WSGI (Web Server Gateway Interface). Он используется при развертывании проекта.

- `asgi.py`: название файла представляет сокращение от Asynchronous Server Gateway Interface и расширяет возможности WSGI, добавляя поддержку для взаимодействия между асинхронными веб-серверами и приложениями.

Запустим проект на выполнение. Для этого с помощью команды `cd` перейдем в консоли к папке проекта. И затем для запуска проекта выполним следующую команду:

```
cd test_project
```

```
(venv) PS C:\Users\Vital\Python\Test2> cd test_project
(venv) PS C:\Users\Vital\Python\Test2\test_project>
```

```
python manage.py runserver
```

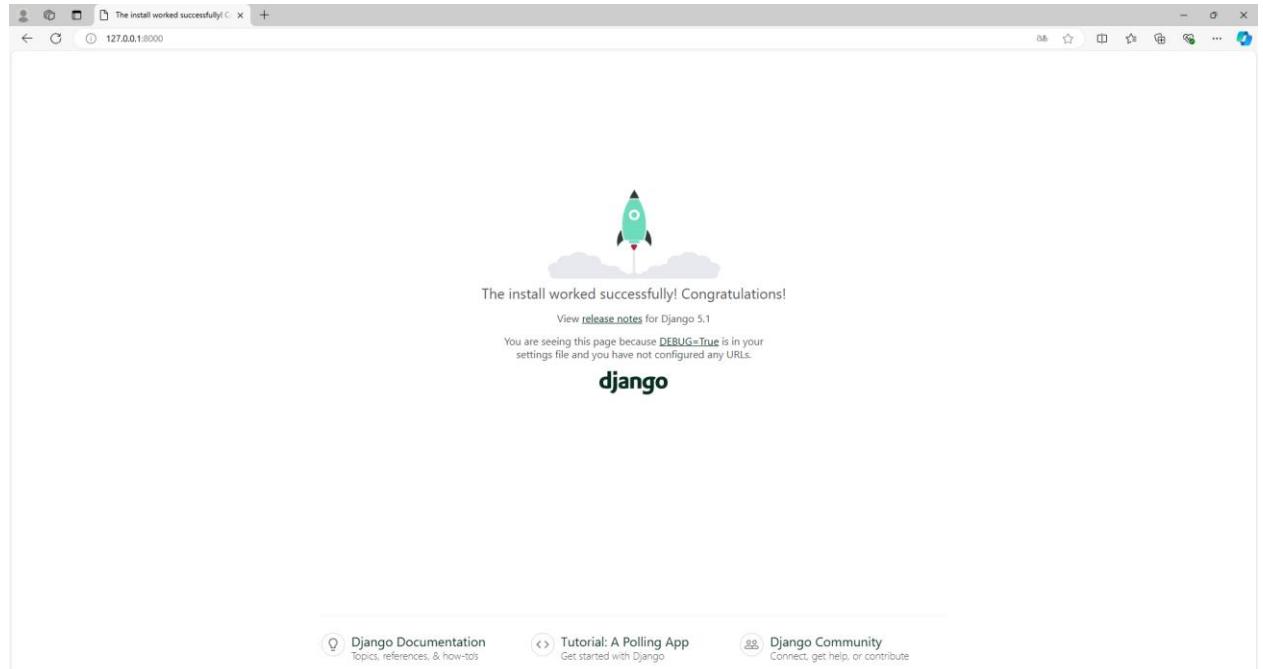
```
(venv) PS C:\Users\Vital\Python\Test2\test_project> python manage.py runserver
```

```
(venv) PS C:\Users\Vital\Python\Test2\test_project> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
November 07, 2024 - 12:33:51
Django version 5.1.3, using settings 'test_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

[07/Nov/2024 12:34:56] "GET / HTTP/1.1" 200 12068
Not Found: /favicon.ico
[07/Nov/2024 12:34:56] "GET /favicon.ico HTTP/1.1" 404 2214
[07/Nov/2024 12:36:07] "GET / HTTP/1.1" 200 12068
Not Found: /favicon.ico
[07/Nov/2024 12:36:07] "GET /favicon.ico HTTP/1.1" 404 2214
```



## Создание первого приложения

Веб-приложение или проект Django состоит из отдельных приложений. Вместе они образуют полноценное веб-приложение. Каждое приложение представляет какую-то определенную функциональность или группу функциональности. Один проект может включать множество приложений. Это позволяет выделить группу задач в отдельный модуль и разрабатывать их относительно независимо от других. Кроме того, мы можем переносить приложение из одного проекта в другой независимо от другой функциональности проекта.

При создании проекта он уже содержит несколько приложений по умолчанию.

- `django.contrib.admin`
- `django.contrib.auth`
- `django.contrib.contenttypes`
- `django.contrib.sessions`
- `django.contrib.messages`
- `django.contrib.staticfiles`

Список всех приложений можно найти в проекте в файле `settings.py` в переменной `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
]
```

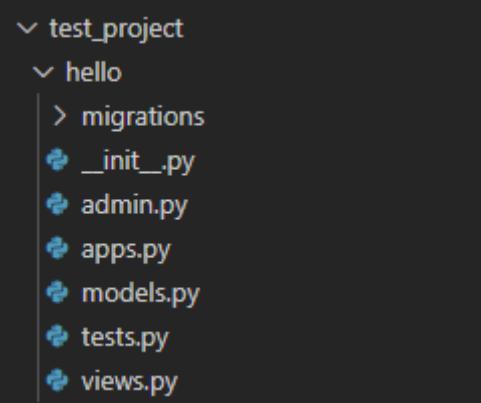
И, конечно, мы можем создавать свои приложения, которые реализуют определенный функционал. Для создания приложения в проекте применяется команда

```
python manage.py startapp название_приложения
```

Так, создадим в проекте, созданном в прошлой теме или новом, первое приложение, которое будет называться `hello`. Для этого выполним в командной строке/терминале следующую команду:

```
python manage.py startapp hello
```

```
• (venv) PS C:\Users\Vital\Python\Test2\test_project> python manage.py startapp hello
```



Рассмотрим вкратце его структуру:

- папка **migrations**: предназначена для хранения миграций - скриптов, которые позволяют синхронизировать структуру базы данных с определением моделей
- **\_\_init\_\_.py**: указывает интерпретатору python, что текущий каталог будет рассматриваться в качестве пакета
- **admin.py**: предназначен для административных функций, в частности, здесь производится регистрация моделей, которые используются в интерфейсе администратора
- **apps.py**: определяет конфигурацию приложения
- **models.py**: хранит определение моделей, которые описывают используемые в приложении данные
- **tests.py**: хранит тесты приложения
- **views.py**: определяет функции, которые получают запросы пользователей, обрабатывают их и возвращают ответ

Но пока приложение никак не задействуется. Его надо зарегистрировать в проекте Django. Для этого откроем файл **settings.py** и добавим в конец массива **INSTALLED\_APPS** наше приложение:

```
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'hello',
41 ]
```

The screenshot shows the PyCharm IDE interface. On the left, the Project tool window displays a hierarchical view of the project structure. It includes a 'test\_project' folder containing a 'hello' application with files like \_\_init\_\_.py, admin.py, apps.py, models.py, tests.py, and views.py. Below 'test\_project' are \_\_pycache\_\_, \_\_init\_\_.py, asgi.py, urls.py, wsgi.py, db.sqlite3, manage.py, venv, and main.py. A yellow arrow points from the 'hello' application folder towards the 'INSTALLED\_APPS' section of the settings.py file. The settings.py file itself is open in the main editor window, showing code for allowed hosts, application definition, installed apps (which currently lists 'django.contrib.admin', 'django.contrib.auth', 'django.contrib.contenttypes', 'django.contrib.sessions', 'django.contrib.messages', 'django.contrib.staticfiles', and 'hello'), and middleware.

```
27  
28 ALLOWED_HOSTS = []  
29  
30  
31 # Application definition  
32  
33 INSTALLED_APPS = [  
34     'django.contrib.admin',  
35     'django.contrib.auth',  
36     'django.contrib.contenttypes',  
37     'django.contrib.sessions',  
38     'django.contrib.messages',  
39     'django.contrib.staticfiles',  
40     'hello',  
41 ]  
42  
43 MIDDLEWARE = [  
44     'django.middleware.security.SecurityMiddleware',  
45     'django.contrib.sessions.middleware.SessionMiddleware',  
46     'django.middleware.common.CommonMiddleware',
```

В проекте может быть несколько приложений, и каждое из них надо добавлять таким вот образом.

Теперь определим какие-нибудь простейшие действия, которые будет выполнять данное приложение, например, отправлять в ответ пользователю строку "Hello World".

Для этого перейдем в проекте приложения **hello** к файлу **views.py**, который по умолчанию должен выглядеть следующим образом:

```
1 from django.shortcuts import render  
2  
3 # Create your views here.  
4
```

Изменим код следующим образом:

```
1 from django.http import HttpResponse  
2  
3 def index(request):  
4     return HttpResponse("Hello World")
```

В данном случае мы импортируем класс **HttpResponse** из стандартного пакета *django.http*. Затем определяется функция **index()**, которая в качестве параметра получает объект запроса **request**. Класс **HttpResponse** предназначен для создания ответа, который отправляется пользователю. И с помощью выражения `return HttpResponse("Hello World")` мы отправляем пользователю строку "Hello World"

Теперь также в основном проекте Django откроем файл **urls.py**, который позволяет сопоставить маршруты с представлениями, которые будут обрабатывать запрос по этим маршрутам. По умолчанию этот файл выглядит следующим образом:

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Первой строкой из модуля *django.contrib* импортируется класс AdminSite, который предоставляет возможности работы с интерфейсом администратора. Второй строкой из модуля *django.urls* импортируется функция **path**. Эта функция задает сопоставление определенного маршрута с функцией обработки. Так, в данном случае маршрут "admin/" будет обрабатываться методом *admin.site.urls*.

Но выше мы определили функцию *index* в файле **views.py**, который возвращает пользователю строку "Hello World". Поэтому изменим файл **urls.py** следующим образом:

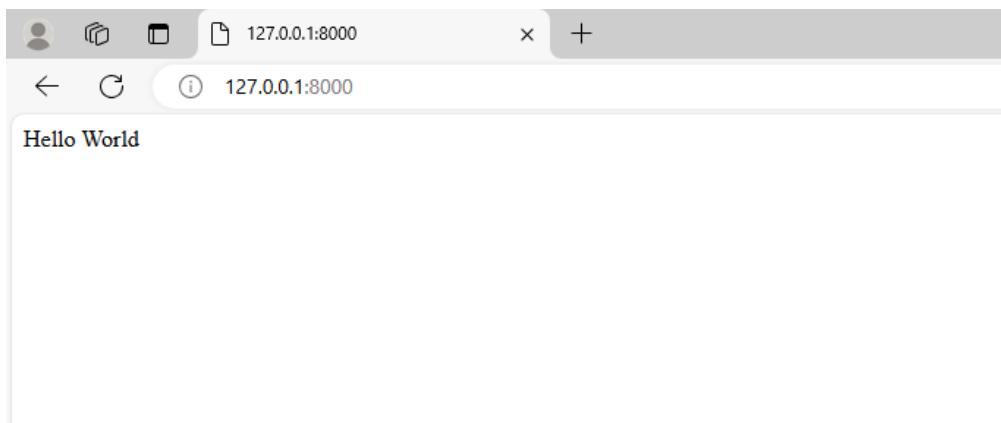
```
from django.urls import path
from hello import views

urlpatterns = [
    path('', views.index, name='home'),
]
```

Теперь снова запустим приложение командой

```
python manage.py runserver
```

И вновь перейдем в браузере по адресу <http://127.0.0.1:8000/>, и браузер нам отобразит строку "Hello World":



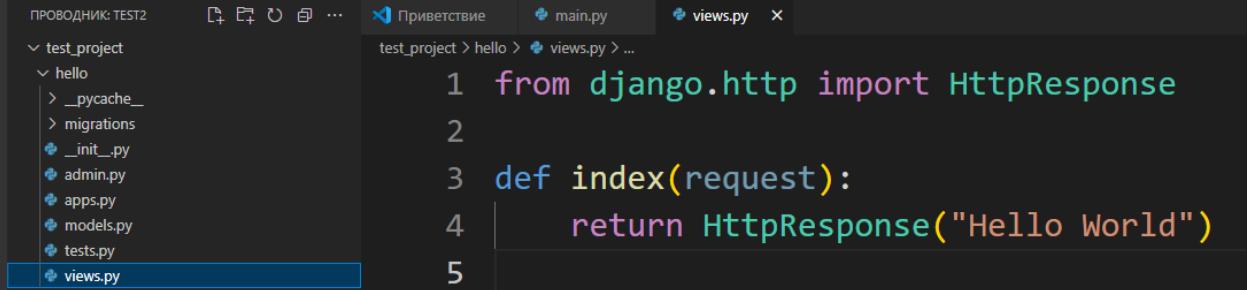
## Представления и маршрутизация

### Обработка запроса

Центральным моментом любого веб-приложения является обработка запроса, который отправляет пользователь. В Django за обработку запроса отвечают представления или **views**. По сути представления представляют функции обработки, которые принимают данные запроса в виде объекта **HttpRequest** из пакета django.http и генерируют некоторый результат, который затем отправляется пользователю.

По умолчанию представления размещаются в приложении в файле **views.py**.

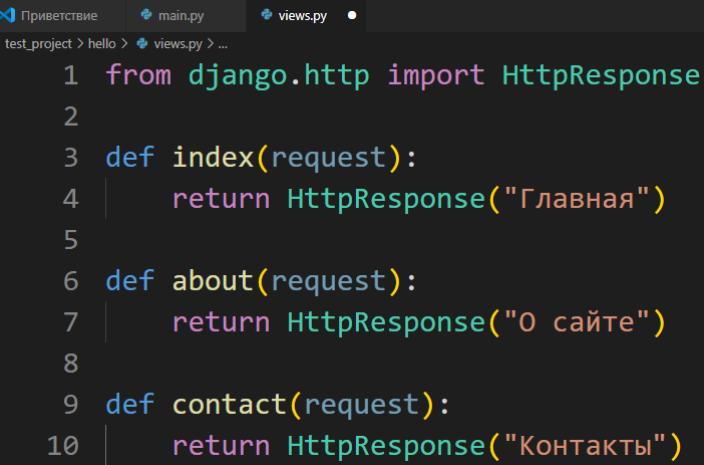
Например, возьмем стандартный проект, в который добавлено приложение (например, проект созданный в прошлой главе



The screenshot shows the PyCharm IDE interface. On the left, the project tree displays a 'test\_project' folder containing a 'hello' app with files like \_\_init\_\_.py, admin.py, apps.py, models.py, tests.py, and views.py. The 'views.py' file is selected and shown in the main code editor. The code in 'views.py' is:

```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("Hello World")
5
```

изменим файл **views.py** следующим образом:

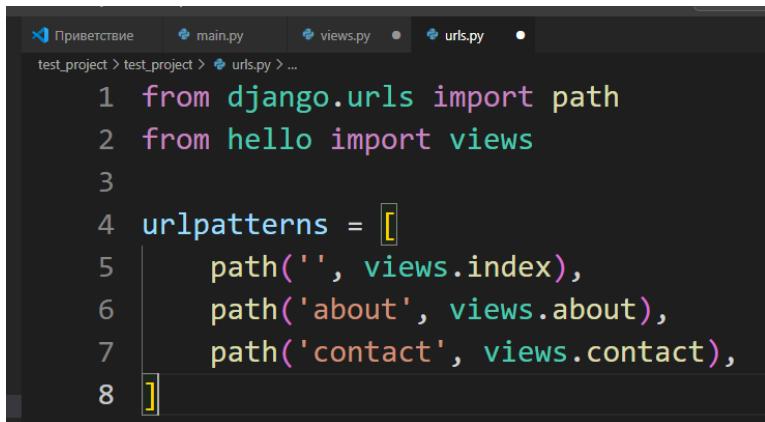


The screenshot shows the PyCharm IDE interface with the 'views.py' file open. The code has been modified to include three functions: index, about, and contact. The code is:

```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("Главная")
5
6 def about(request):
7     return HttpResponse("О сайте")
8
9 def contact(request):
10    return HttpResponse("Контакты")
```

В данном случае определены три функции, которые будут обрабатывать запросы. Каждая функция принимает в качестве параметра `request` объект `HttpRequest`, который хранит информацию о запросе. Однако в данном случае она нам не нужны, поэтому параметр никак не используется. Для генерации ответа в конструктор объекта `HttpResponse` передается некоторая строка. Это может быть в том числе и код html в виде строки.

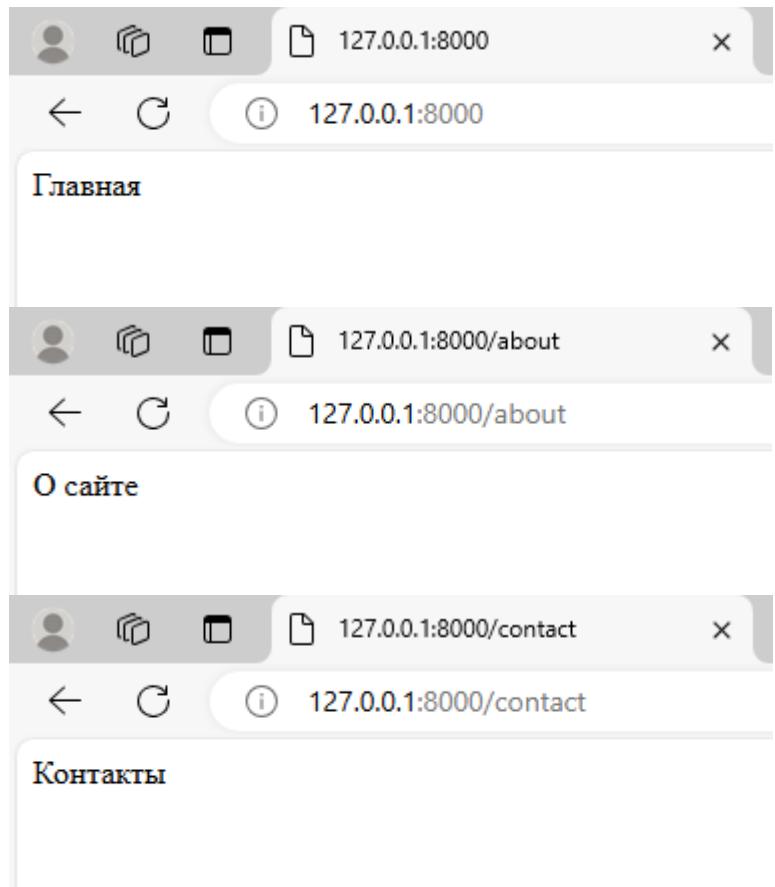
Чтобы эти функции сопоставлялись с запросами, надо определить для них маршруты в проекте в файле urls.py. В частности, изменим этот файл следующим образом:



```
Приветствие main.py views.py urls.py
test_project > test_project > urls.py > ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path('', views.index),
6     path('about', views.about),
7     path('contact', views.contact),
8 ]
```

Переменная **urlpatterns** определяет набор сопоставлений функций обработки с определенными строками запроса. Например, запрос к корню веб-сайта будет обрабатываться функцией index, запрос по адресу "about" будет обрабатываться функцией about, а запрос "contact" - функцией contact.

Запустим проект и обратимся по некоторым из этих адресов.



При этом мы можем отправлять не простой текст, а, например, код html, который затем интерпретируется браузером. Так, изменим файл views.py следующим образом:

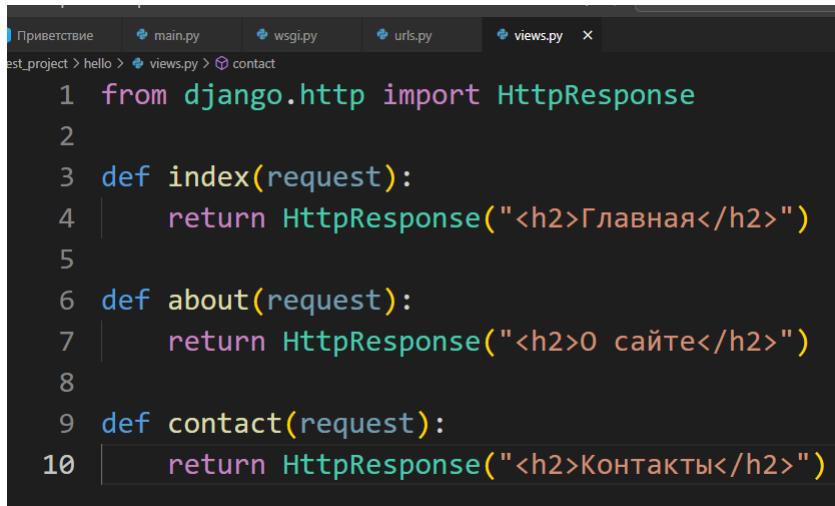
```
Приветствие main.py wsgi.py urls.py views.py x
est_project > hello > views.py > contact
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h2>Главная</h2>")
5
6 def about(request):
7     return HttpResponseRedirect("<h2>О сайте</h2>")
8
9 def contact(request):
10    return HttpResponseRedirect("<h2>Контакты</h2>")
```

The screenshot shows a web browser window with three tabs, each displaying a different page from a Django application running at `127.0.0.1:8000`.

- Tab 1:** URL `127.0.0.1:8000`. Content: **Главная** (Main).
- Tab 2:** URL `127.0.0.1:8000/contact`. Content: **Контакты** (Contacts).
- Tab 3:** URL `127.0.0.1:8000/about`. Content: **О сайте** (About).

## Определение маршрутов и функции path и re\_path

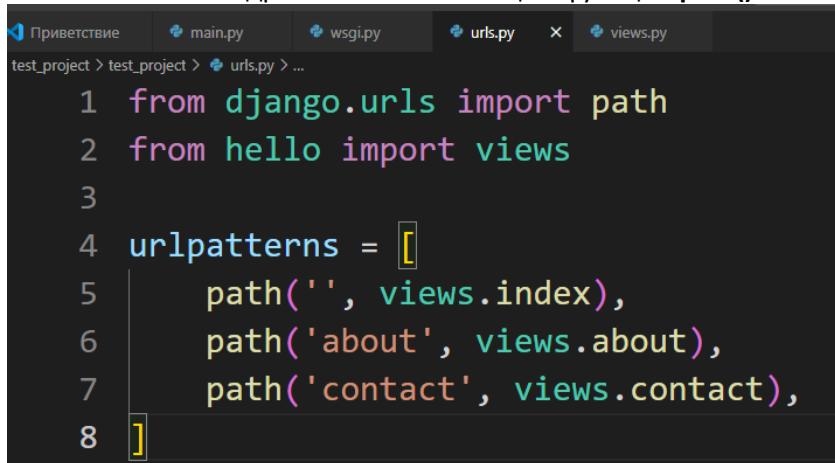
В прошлой теме рассматривалось сопоставление адресов URL и функций, которые обрабатывают запросы по этим адресам. Например, у нас есть следующие функции в файле **views.py**:



```
Приветствие main.py wsgi.py urls.py views.py

1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h2>Главная</h2>")
5
6 def about(request):
7     return HttpResponseRedirect("<h2>О сайте</h2>")
8
9 def contact(request):
10    return HttpResponseRedirect("<h2>Контакты</h2>")
```

Это так называемые функции-представления или view function. И в файле **urls.py** проекта они сопоставляются с адресами URL с помощью функции **path()**:



```
Приветствие main.py wsgi.py urls.py views.py

1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path('', views.index),
6     path('about', views.about),
7     path('contact', views.contact),
8 ]
```

За сопоставление путей и функций-представлений отвечает функция **path()**, которая располагается в пакете **django.urls** и которая принимает четыре параметра:

```
path(route, view, kwargs=None, name=None)
```

- **route**: представляет шаблон адреса URL, которому должен соответствовать запрос
- **view**: функция-представление, которое обрабатывает запрос
- **kwargs**: дополнительные аргументы, которые передаются в функцию-представление
- **name**: название маршрута

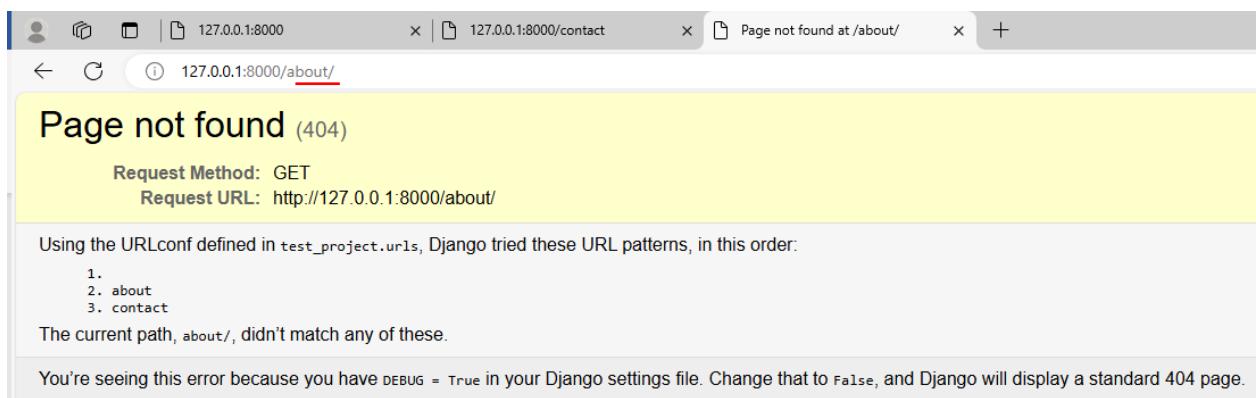
В примере выше применялись только первые два параметра, которые являются обязательными: запрошенный адрес URL и функция, которая обрабатывает запрос по этому адресу. Дополнительно через третий параметр можно указать имя маршрута:

```
path('', views.index, name='home'),
```

В данном случае маршрут будет называться "home".

## re\_path

Хотя мы можем успешно применять функцию `path()` для определения маршрутов, она довольно ограничена по своему действию. Запрошенный путь должен в точности соответствовать указанному в маршруте адресу URL. Так, в примере выше чтобы функция `views.about` могла обрабатывать запрос, адрес должен быть в точности "about". Например, стоит нам указать слеш в конце: "about/" и django уже не сможет сопоставить путь с запросом.



В качестве альтернативы для определения маршрутов мы можем использовать функцию `re_path()`, которая также располагается в пакете `django.urls` и имеет тот же набор параметров:

```
re_path(route, view, kwargs=None, name=None)
```

Ее преимущество состоит в том, что она позволяет задать адреса URL с помощью регулярных выражений.

Например, изменим определение файла `urls.py` следующим образом:

A screenshot of a code editor showing a Python file named 'urls.py'. The file content is as follows:

```
from django.urls import path, re_path
from hello import views

urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
]
```

The code editor interface shows tabs for 'Приветствие', 'main.py', 'wsgi.py', 'urls.py', and 'views.py'. The 'urls.py' tab is active.

Адрес в первом маршруте по-прежнему образуется с помощью функции path и указывает на корень веб-приложения.

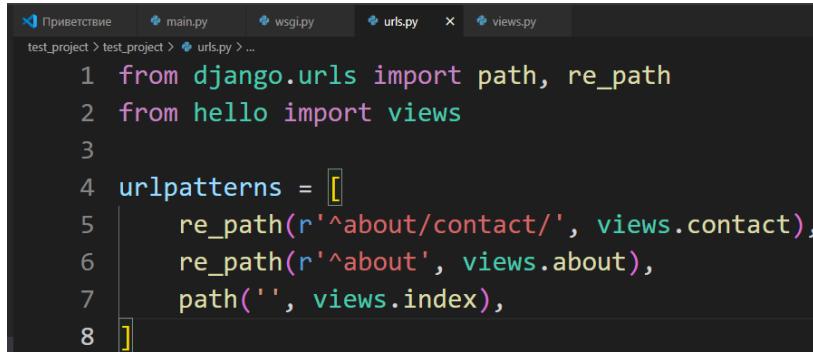
Остальные два маршрута образуются с помощью функции re\_path(). Причем, поскольку определяется регулярное выражение, то перед строкой с шаблоном адреса URL ставится буква r. В самом шаблоне адреса можно использовать различные элементы синтаксиса регулярных выражений. В частности, выражение ^about указывает, что адрес должен начинаться с "about". Однако он необязательно в точности должен соответствовать строке "about", как это было в случае с функцией path.

Например, мы можем обратиться по любому адресу, главное чтобы он начинался с "about", и тогда подобный запрос будет обрабатываться функцией views.about.



## Очередность маршрутов

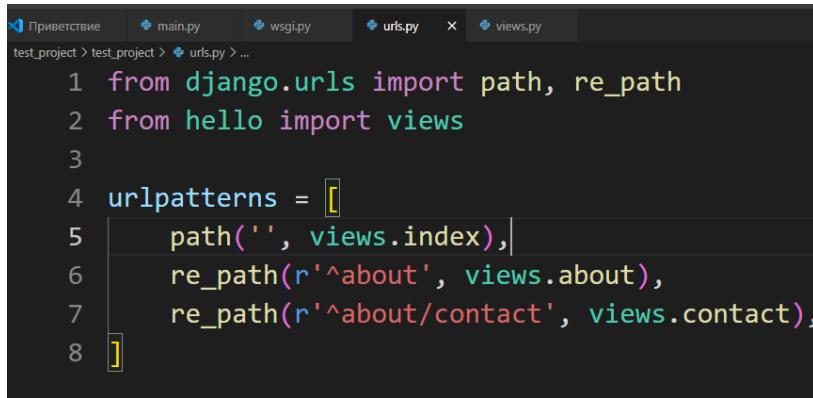
Когда запрос приходит к приложению, то система проверяет соответствие запроса маршрутам по мере их определения: вначале сравнивается первый маршрут, если он не подходит, то сравнивается второй и так далее. Поэтому более общие маршруты должны определяться в последнюю очередь, а более конкретные маршруты должны идти в начале. Например:



```
1 from django.urls import path, re_path
2 from hello import views
3
4 urlpatterns = [
5     re_path(r'^about/contact/', views.contact),
6     re_path(r'^about', views.about),
7     path('', views.index),
8 ]
```

В данном случае адрес "`^about/contact`" представляет более конкретный маршрут по сравнению с "`^about`". Поэтому он определяется в первую очередь.

Если бы было наоборот:



```
1 from django.urls import path, re_path
2 from hello import views
3
4 urlpatterns = [
5     path('', views.index),
6     re_path(r'^about', views.about),
7     re_path(r'^about/contact', views.contact),
8 ]
```

то запрос по адресу "about/contact" обрабатывался бы функцией `views.about`

## Основные элементы синтаксиса регулярных выражений

Некоторые базовые элементы регулярных выражений, которые можно использовать для определения адресов URL:

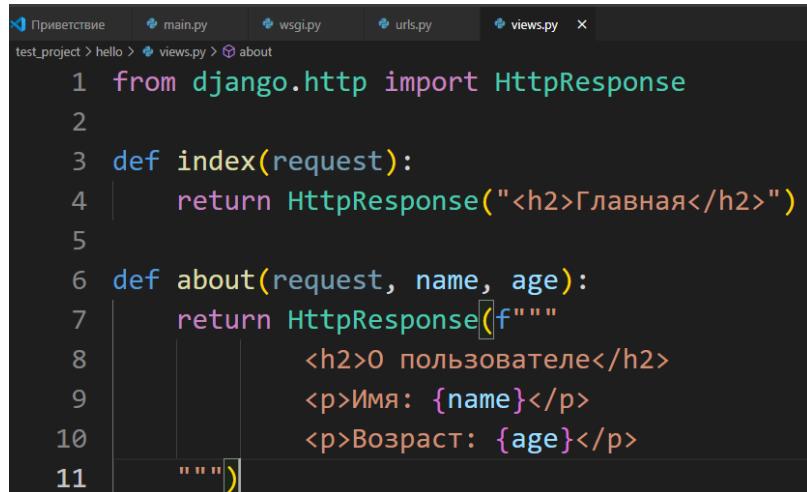
- `^(начало адреса)`
- `$(конец адреса)`
- `+(1 и более символов)`
- `?(0 или 1 символ)`
- `{n}(n символов)`
- `{n, m}(от n до m символов)`
- `.(любой символ)`
- `\d+(одна или несколько цифр)`
- `\D+(одна или несколько НЕ цифр)`
- `\w+(один или несколько буквенных символов)`

Рассмотрим несколько возможных сопоставлений шаблонов адресов и запросов:

Адрес	Запрос
<code>r'^\$'</code>	<code>http://127.0.0.1/</code> (корень сайта)
<code>r'^about'</code>	<code>http://127.0.0.1/about/</code> или <code>http://127.0.0.1/about/contact</code>
<code>r'^about/contact'</code>	<code>http://127.0.0.1/about/contact</code>
<code>r'^products/\d+/'</code>	<code>http://127.0.0.1/products/23/</code> или <code>http://127.0.0.1/products/6459/abc</code> Но не соответствует запросу <code>http://127.0.0.1/products/abc/</code>
<code>r'^products/\D+/'</code>	<code>http://127.0.0.1/products/abc/</code> или <code>http://127.0.0.1/products/abc/123</code> Не соответствует запросу <code>http://127.0.0.1/products/123/</code> или <code>http://127.0.0.1/products/123/abc</code>
<code>r'^products/phones tablets/'</code>	<code>http://127.0.0.1/products/phones/1</code> или <code>http://127.0.0.1/products/tablets/</code> Не соответствует запросу <code>http://127.0.0.1/products/clothes/</code>
<code>r'^products/\w+'</code>	<code>http://127.0.0.1/products/abc/</code> или <code>http://127.0.0.1/products/123/</code> Не соответствует запросу <code>http://127.0.0.1/products/abc-123</code>
<code>r'^products/[-\w]+/'</code>	<code>http://127.0.0.1/products/abc-123</code>
<code>r'^products/[A-Z]{2}/'</code>	<code>http://127.0.0.1/products/RU</code> Не соответствует запросам <code>http://127.0.0.1/products/Ru</code> или <code>http://127.0.0.1/products/RUS</code>

## Передача значений в функцию

Выше были рассмотрены все параметры функций path и re\_path, кроме одного - kwargs, который позволяет передать в функцию-представление некоторые значения. Например, в файле `views.py` определим следующие функции:



```
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h2>Главная</h2>")
5
6 def about(request, name, age):
7     return HttpResponseRedirect(f"""
8         <h2>О пользователе</h2>
9         <p>Имя: {name}</p>
10        <p>Возраст: {age}</p>
11    """)
```

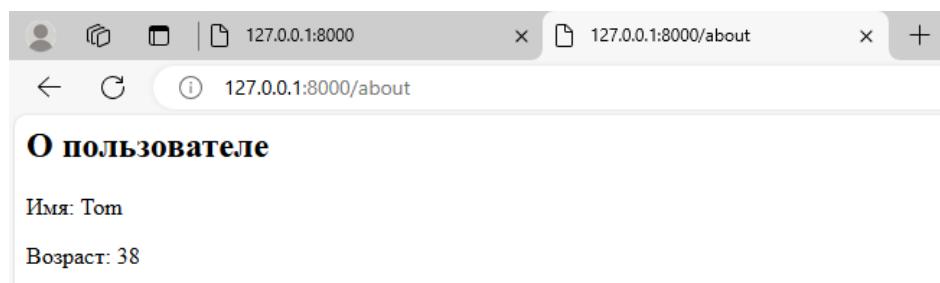
Здесь функция `about()` также принимает два дополнительных параметра: `name` и `age` (условно имя и возраст пользователя). В функции их значения отправляются пользователю вместе с остальным содержимым.

Изменим файл `urls.py`:



```
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path('', views.index),
6     path('about', views.about, kwargs={"name": "Tom", "age": 38}),
7 ]
```

С помощью параметра `kwargs` в функцию `about` передается словарь с двумя значениями - для двух параметров функции. Соответственно при обращении к этой функции мы увидим в браузере соответствующие данные:



## Получение данных запроса. HttpRequest

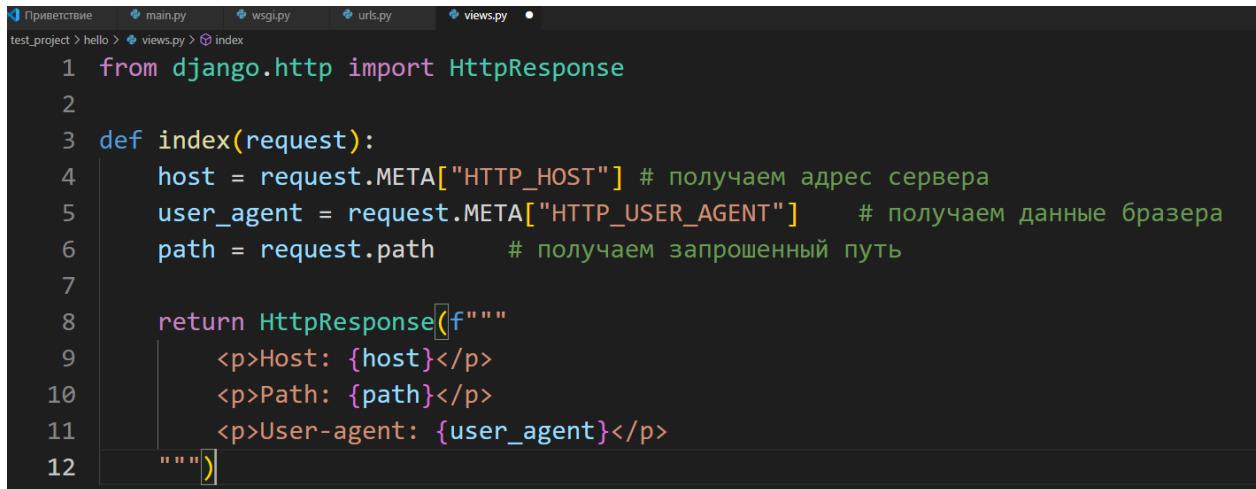
Функции-представления в качестве обязательного параметра получают объект **HttpRequest**, который хранит информацию о запросе. HttpRequest определяет ряд атрибутов, которые хранят информацию о запросе. Выделим следующие из них:

- **scheme**: схема запроса (http или https)
- **body**: представляет тело запроса в виде строки байтов
- **path**: представляет путь запроса
- **method**: метод запроса (GET, POST, PUT и т.д.)
- **encoding**: кодировка
- **content\_type**: тип содержимого запроса (значение заголовка CONTENT\_TYPE)
- **GET**: объект в виде словаря, который содержит параметры запроса GET
- **POST**: объект в виде словаря, который содержит параметры запроса POST
- **COOKIES**: отправленные клиентом куки
- **FILES**: отправленные клиентом файлы
- **META**: хранит все доступные заголовки http в виде словаря. Набор заголовков зависит от клиента и сервера, некоторые из них:
  - CONTENT\_LENGTH: длина содержимого.
  - CONTENT\_TYPE: MIME-тип запроса.
  - HTTP\_ACCEPT: типы ответа, которые принимает клиент.
  - HTTP\_ACCEPT\_ENCODING: кодировка, в которой клиент принимает ответ.
  - HTTP\_ACCEPT\_LANGUAGE: язык ответа, который принимает клиент.
  - HTTP\_HOST: хост сервера.
  - HTTP\_REFERER: страница, с которой клиент отправил запрос (при ее наличии).
  - HTTP\_USER\_AGENT: юзер-агент или информация о браузере клиента.
  - QUERY\_STRING: строка запроса.
  - REMOTE\_ADDR: IP-адрес клиента.
  - REMOTE\_HOST: имя хоста клиента.
  - REMOTE\_USER: аутентификационные данные клиента (при наличии)
  - REQUEST\_METHOD: тип запроса (GET, POST).
  - SERVER\_NAME: имя хоста сервера.
  - SERVER\_PORT: порт сервера.
- **headers**: заголовки запроса в виде словаря

Также HttpRequest определяет ряд методов. Отметим следующие из них:

- **get\_full\_path()**: возвращает полный путь запроса, включая строку запроса
- **get\_host()**: возвращает хост клиента, для этого используется значения заголовков HTTP\_X\_FORWARDED\_HOST (если включена опция USE\_X\_FORWARDED\_HOST) и HTTP\_HOST
- **get\_port()**: возвращает номер порта

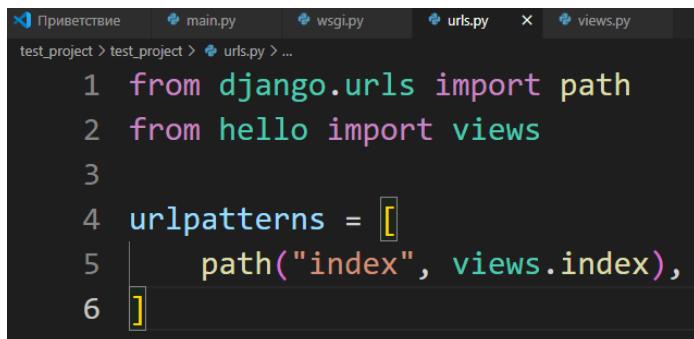
Например, получим некоторую информацию о запросе. Для этого в файле **views.py**:



```
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     host = request.META["HTTP_HOST"] # получаем адрес сервера
5     user_agent = request.META["HTTP_USER_AGENT"] # получаем данные браузера
6     path = request.path # получаем запрошенный путь
7
8     return HttpResponseRedirect(f"""
9         <p>Host: {host}</p>
10        <p>Path: {path}</p>
11        <p>User-agent: {user_agent}</p>
12    """)
```

В данном случае получаем два заголовка "HTTP\_HOST" и "HTTP\_USER\_AGENT" и запрошенный путь.

В файле **urls.py** зарегистрируем данную функцию:



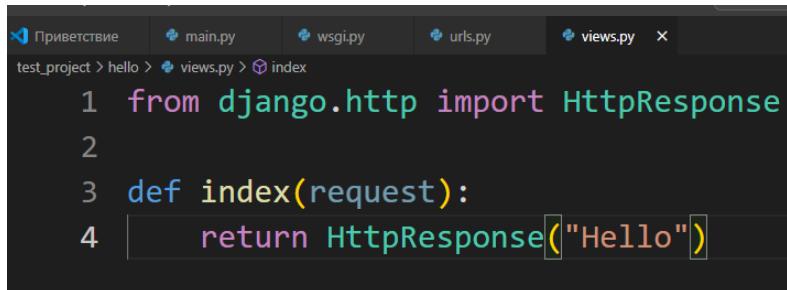
```
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("index", views.index),
6 ]
```

Результат работы:



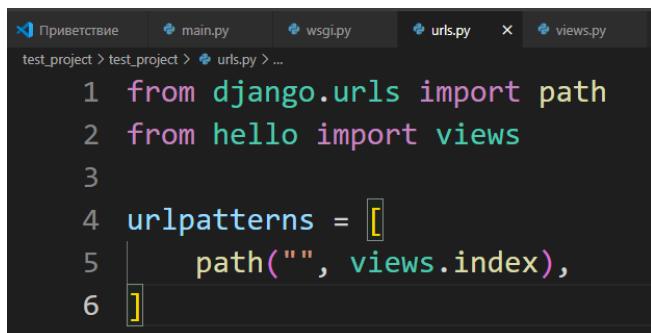
## HttpResponse и отправка ответа

Для отправки ответа клиенту в Django применяется класс **HttpResponse** из пакета **django.http**. В общем случае для отправки некоторых данных достаточно эти данные передать в конструктор HttpResponse. Например, пусть в файле **views.py** имеется простейшая функция-представление, которая отправляет ответ клиенту:

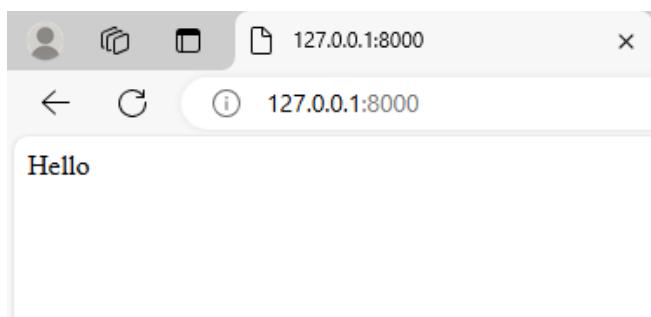


```
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("Hello")
```

И в файле **urls.py** эта функция соотносится с некоторым маршрутом:



```
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6 ]
```



Подобная функция просто передает в HttpResponse некоторый текст, который пользователь затем увидит в браузере. Однако подобной функциональностью HttpResponse не ограничивается. Так, функция инициализации класса определяет несколько параметров:

```
HttpResponse.__init__(content=b'', content_type=None, status=200, reason=None,
charset=None, headers=None)
```

Параметры:

- **content**: содержимое ответа в виде строки байтов. Если передается другое содержимое, то оно конвертируется в строку байтов
- **content\_type**: MIME-тип ответа, устанавливает HTTP-заголовок Content-Type. Если этот параметр не установлен, то применяется mime-тип text/html и значение настройки DEFAULT\_CHARSET, то есть в итоге будет: "text/html; charset=utf-8".

- **charset**: кодировка ответа в виде строки. По умолчанию django пытается установить кодировку из параметра `content_type`, а в случае неудачи для установки кодировки применяется настройка `DEFAULT_CHARSET`.
- **status**: статусный код ответа. По умолчанию равно 200
- **reason\_phrase**: сообщение, которое отправляется в вместе со статусным кодом
- **headers**: заголовки ответа в виде словаря

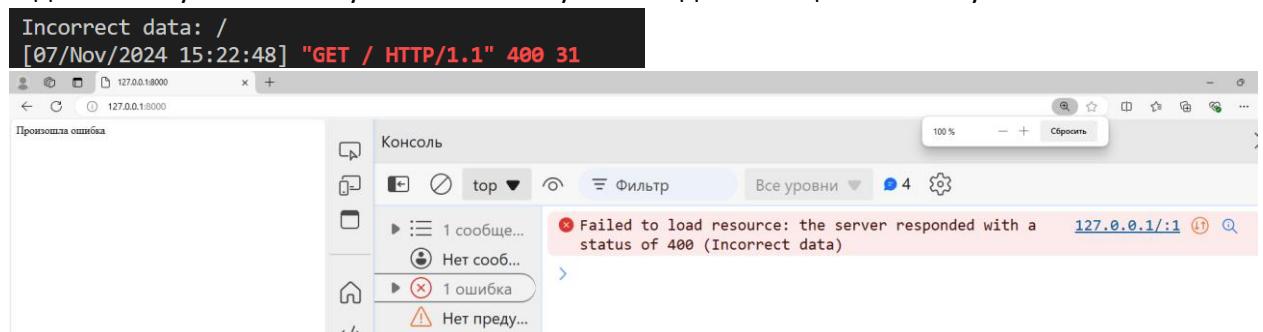
Для хранения отправляемых данных он определяет ряд атрибутов. Некоторые из них:

- **content**: содержимое ответа в виде строки байтов
- **headers**: отправляемые заголовки в виде словаря
- **charset**: кодировка ответа в виде строки. По умолчанию django пытается установить кодировку из заголовка `content_type`, а в случае неудачи для установки кодировки применяется настройка `DEFAULT_CHARSET`.
- **status\_code**: статусный код ответа
- **reason\_phrase**: сообщение, которое отправляется в вместе со статусным кодом

Рассмотрим некоторые возможности. Например, изменим определение функции в файле `views.py`:

```
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("Произошла ошибка", status=400, reason="Incorrect data")
```

В данном случае можно установить статусный код и сообщение к нему:



Установка содержимого и кодировки:

```
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h1>Hello</h1>", content_type="text/plain", charset="utf-8")
```

Хотя в содержимом ответа применяются теги html (<h1>), но браузере теперь будет рассматривать это содержимое как простой текст, потому что установлен заголовок "text/plain":



## Параметры представлений

Функции-представления могут принимать параметры, через которые могут передаваться различные данные. Подобные параметры передаются в адресе URL. Например, в запросе

```
http://127.0.0.1:8000/index/Tom/38/
```

последние два сегмента Tom/38/ могут представлять параметры URL, которые могут быть связаны с параметрами функции-представления через систему маршрутизации.

Подобные параметры еще можно назвать **параметрами маршрута**

## Определение параметров через функцию path

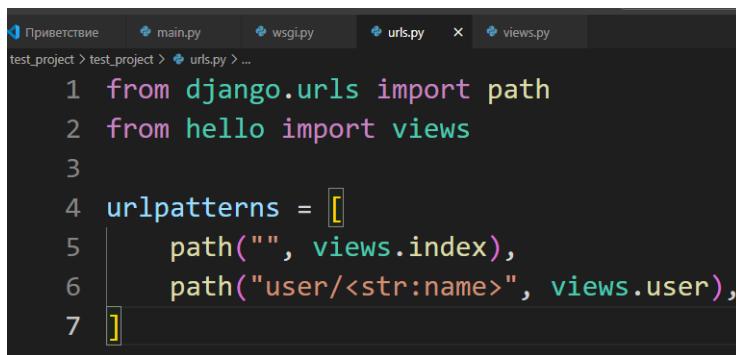
Определим в файле **views.py** следующие функции:



```
Приветствие main.py wsgi.py urls.py views.py
t_project > hello > views.py > user
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h2>Главная</h2>")
5
6 def user(request, name):
7     return HttpResponseRedirect(f"<h2>Имя: {name}</h2>")
```

В данном случае функция user имеет два параметра. Для второго параметра - name мы будем получать данные из строки запроса. То есть это будет параметр маршрута. И как любой другой параметр мы сможем использовать его внутри функции.

Далее в файле **urls.py** определим следующий код:



```
Приветствие main.py wsgi.py urls.py views.py
test_project > test_project > urls.py > ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6     path("user/<str:name>", views.user),
7 ]
```

Для второго маршрута здесь определен параметр name (который соответствует параметру name функции views.user). Параметры заключаются в угловые скобки в формате <спецификатор: название\_параметра>. Например, здесь параметр name имеет спецификатор str.

По умолчанию Django предоставляет следующие спецификаторы:

- **str**: соответствует любой строке за исключением символа "/". Если спецификатор не указан, то используется по умолчанию
- **int**: соответствует любому положительному числу

- **slug**: соответствует последовательности буквенных символов ASCII, цифр, дефиса и символа подчеркивания, например, `building-your-1st-django-site`
- **uuid**: соответствует идентификатору UUID, например, `075194d3-6885-417e-a8a8-6c931e272f00`
- **path**: соответствует любой строке, которая также может включать символ "/" в отличие от спецификатора str

Также отмечу, что количество и название параметров в шаблонах адресов URL соответствуют количеству и названиям параметров соответствующих функций, которые обрабатывают запросы по данным адресам.

Запустим приложение, обратимся к функции `views.user`, например, с помощью запроса `http://127.0.0.1:8000/user/Tom`, и через строку запроса - через ее третий сегмент мы сможем передать значение для параметра name



Подобным образом можно определить и большее количество параметров. Например, добавим второй параметр в функцию user в `views.py`:

```
риветствие main.py wsgi.py urls.py views.py
project > hello > views.py > user
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("<h2>Главная</h2>")
5
6 def user(request, name, age):
7     return HttpResponse(f"<h2>Имя: {name} Возраст:{age}</h2>")
```

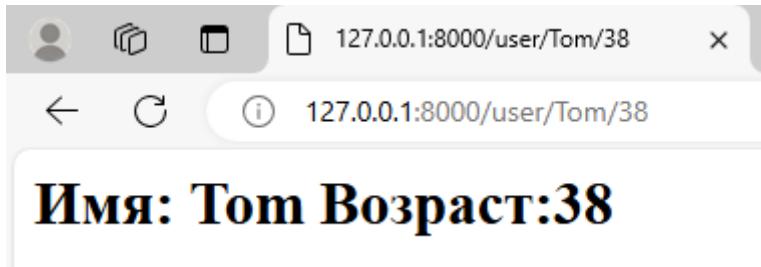
А в файле `urls.py` добавим в маршрут параметр age:

```
риветствие main.py wsgi.py urls.py views.py
project > test_project > urls.py ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6     path("user/<name>/<int:age>", views.user),
7 ]
```

Предполагается, что параметр age будет представлять число, поэтому для него используется спецификатор `int`. В этом случае мы можем обратиться к функции user, например, с помощью запроса

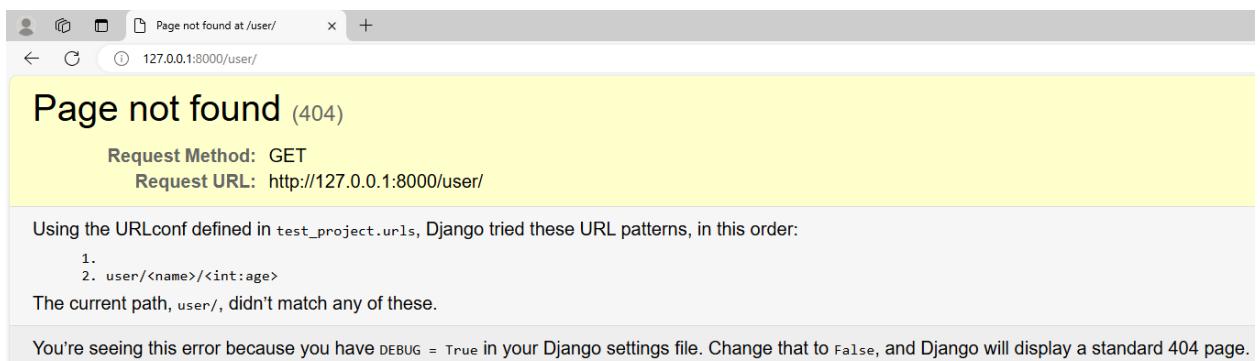
```
http://127.0.0.1:8000/user/Tom/38
```

В этом случае сегмент "/Tom/" будет сопоставлен с параметром name, а "/38" - с параметром age.



### Значения для параметров по умолчанию

В примере выше использовались два параметра, но что, если мы не передадим для одного или обоих параметров значения?



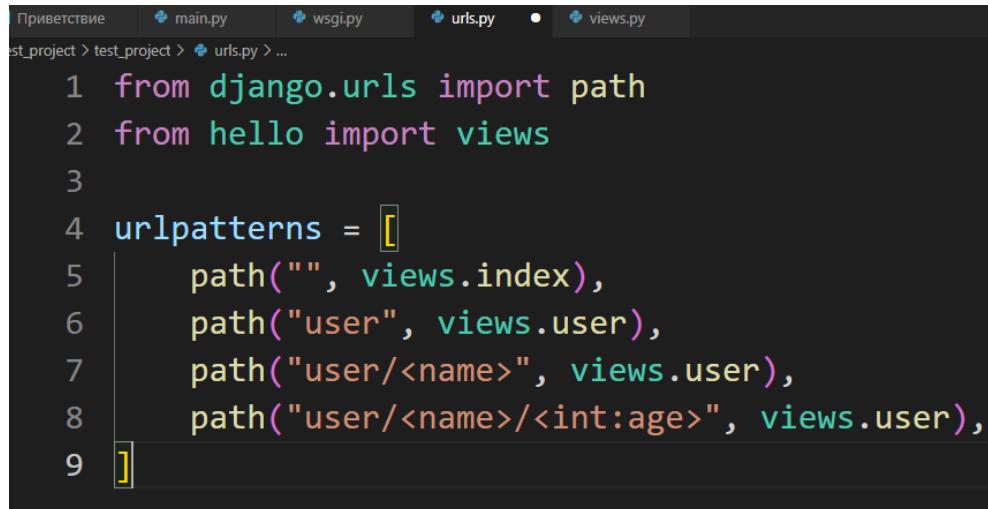
В этом случае мы получим ошибку. Параметры маршрута являются частью шаблона URL. Соответственно если в строке запроса не передаются значения для параметров, такой запрос не соответствует маршруту. Поэтому Django не сможет найти нужный маршрут для обработки запроса, и мы получим ошибку 404 (ресурс не найден).

Однако такое поведение не всегда может быть желательным. И мы можем задать для параметров маршрута значения по умолчанию на случай, если через строку запроса не передаются значения. Так, для функции user в `views.py` определим значения для параметров по умолчанию:

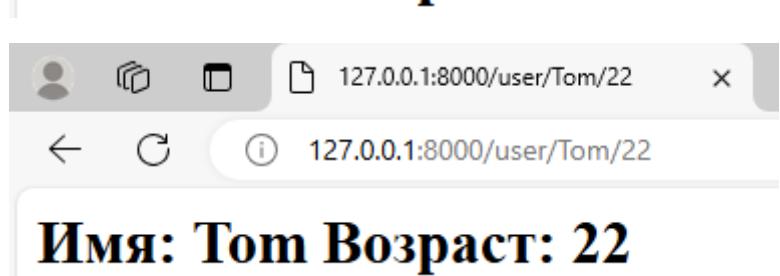
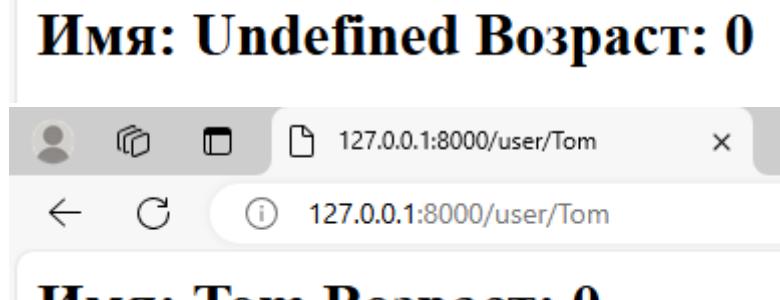
```
риветствие main.py wsgi.py urls.py views.py ×
.project > hello > views.py > user
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("<h2>Главная</h2>")
5
6 def user(request, name="Undefined", age =0):
7     return HttpResponse(f"<h2>Имя: {name} Возраст: {age}</h2>")
```

В данном случае, если для параметра name не передается значение, то он получает в качестве значения строку "Undefined". Для параметра age значение по умолчанию 0.

В этом случае для функции user в файле `urls.py` надо определить дополнительные маршруты, которые не учитывают необязательные параметры:



```
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6     path("user", views.user),
7     path("user/<name>", views.user),
8     path("user/<name>/<int:age>", views.user),
9 ]
```



## Определение параметров через функцию re\_path

Подобным образом мы можем использовать функцию `re_path` для определения параметров. Определим в приложении в файле `views.py` следующие функции:

```
известствие main.py wsgi.py urls.py views.py
project > hello > views.py > ...
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h2>Главная</h2>")
5
6 def user(request, name, age):
7     return HttpResponseRedirect(f"<h2>Имя: {name} Возраст: {age}</h2>")
```

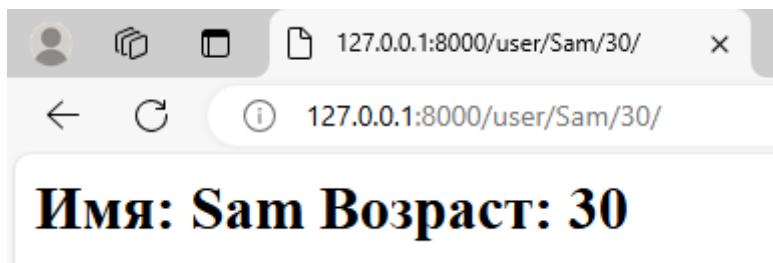
Теперь изменим файл `urls.py`, чтобы он мог сопоставить данные функции с запросами:

```
известствие main.py wsgi.py urls.py views.py
project > test_project > urls.py > ...
1 from django.urls import path, re_path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6     re_path(r"^user/(?P<name>\D+)/(?P<age>\d+)", views.user),
7 ]
```

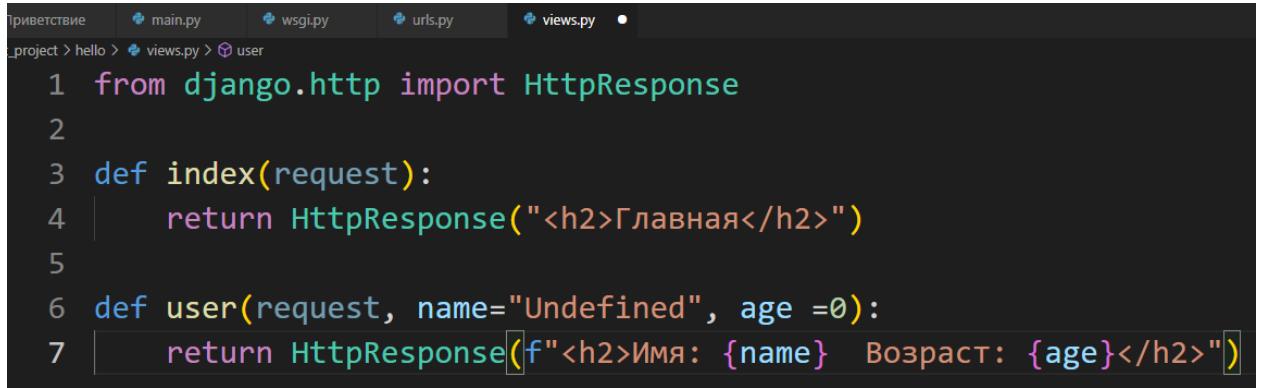
Для представления параметра в шаблоне адреса используется выражение `?P<>`. Общее определение параметру соответствует формату (`?P<имя_параметра>регулярное_выражение`). Между угловыми скобками помещается название параметра. После закрывающей угловой скобки идет регулярное выражение, которому должно соответствовать значение параметра.

Во втором шаблоне адреса определяются два параметра: `name` и `age`. При этом параметр `age` должен представлять число, а параметр `name` должен состоять только из буквенных символов.

Теперь мы можем через адресную строку передать данные в приложение:



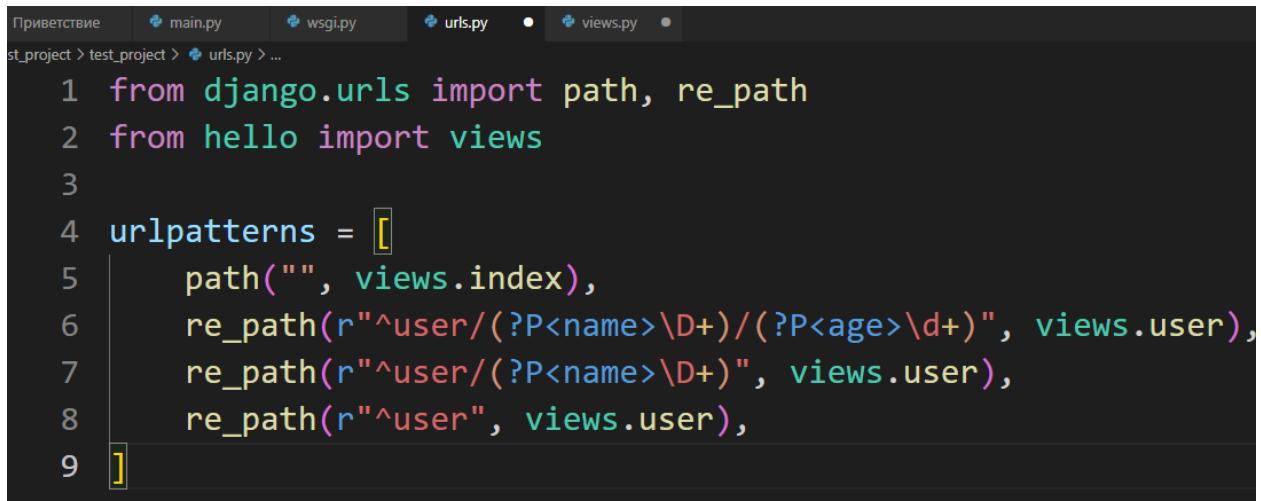
Также мы можем указать для определенных параметров значения по умолчанию:



```
Приветствие main.py wsgi.py urls.py views.py

.project > hello > views.py > user
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h2>Главная</h2>")
5
6 def user(request, name="Undefined", age=0):
7     return HttpResponseRedirect(f"<h2>Имя: {name} Возраст: {age}</h2>")
```

В этом случае надо дополнительно определить еще маршруты в файле `urls.py` для тех запросов, в которых не передаются значения для маршрутов:



```
Приветствие main.py wsgi.py urls.py views.py

st_project > test_project > urls.py > ...
1 from django.urls import path, re_path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6     re_path(r"^user/(?P<name>\D+)/(?P<age>\d+)", views.user),
7     re_path(r"^user/(?P<name>\D+)", views.user),
8     re_path(r"^user", views.user),
9 ]
```

Обратите внимание на порядок размещения маршрутов: в отличие от случая с функцией `path` теперь сначала размещаются более конкретные маршруты с большим количеством параметров.

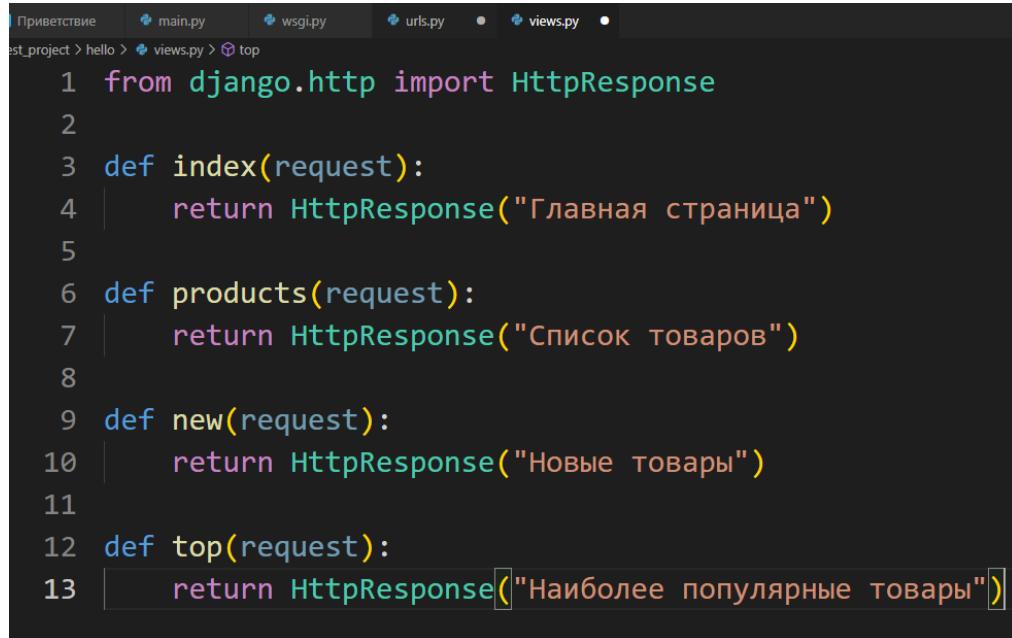
## Вложенные маршруты и функция include

Функция **include()** позволяет определить вложенные маршруты или подмаршруты для некоторого маршрута. В качестве параметра она принимает набор маршрутов:

```
include(pattern_list)
```

Параметр **pattern\_list** представляет набор вызовов функций **path()** и/или **re\_path()**.

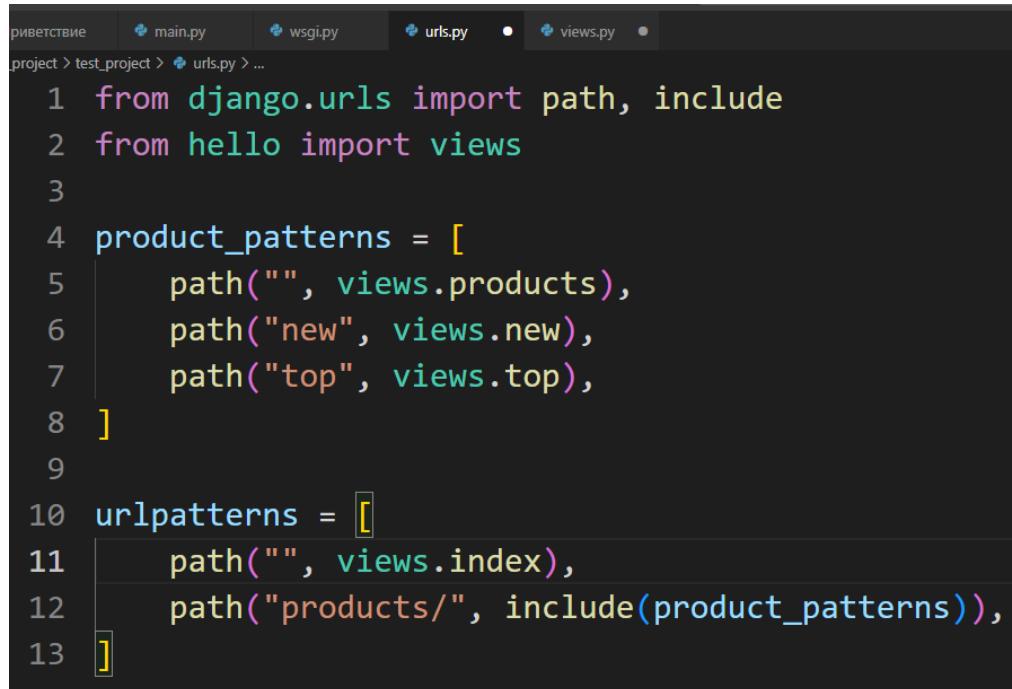
Например, определим в файле **views.py** следующие функции:



```
Приветствие main.py wsgi.py urls.py views.py top
test_project > hello > views.py > top
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("Главная страница")
5
6 def products(request):
7     return HttpResponse("Список товаров")
8
9 def new(request):
10    return HttpResponse("Новые товары")
11
12 def top(request):
13     return HttpResponse("Наиболее популярные товары")
```

Здесь последние три функции функционально относятся к товарам.

И определим в файле **urls.py** следующие маршруты:



```
Приветствие main.py wsgi.py urls.py views.py top
project > test_project > urls.py > ...
1 from django.urls import path, include
2 from hello import views
3
4 product_patterns = [
5     path("", views.products),
6     path("new", views.new),
7     path("top", views.top),
8 ]
9
10 urlpatterns = [
11     path("", views.index),
12     path("products/", include(product_patterns)),
13 ]
```

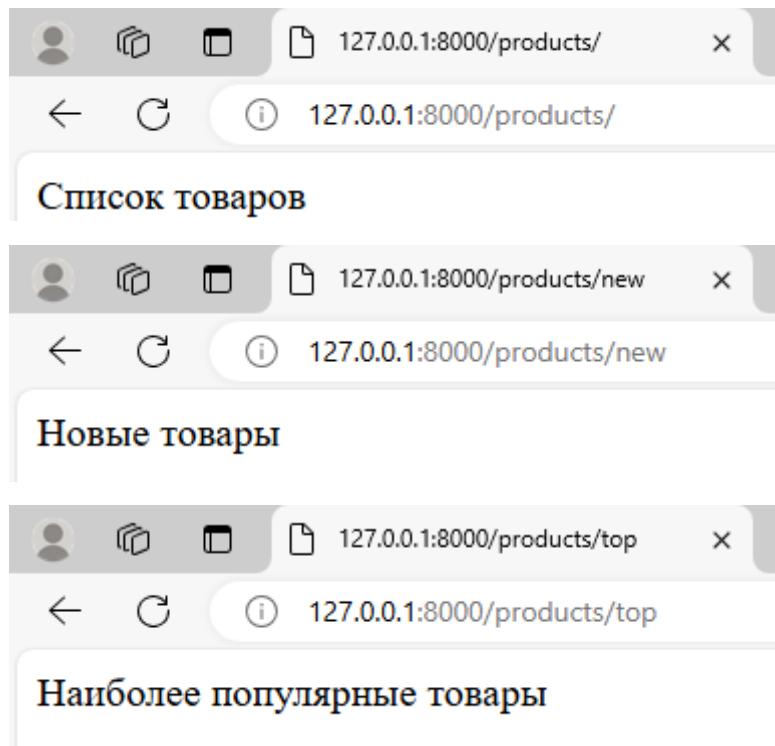
Здесь в виде переменной `product_patterns` отдельно определен набор маршрутов, который касается товаров.

Для установки этих маршрутов этот список передается в функцию `include()`

```
path("products", include(product_patterns)),
```

Причем этот список будет ассоциирован с шаблоном "product". В этом случае шаблоны вложенных маршрутов будут объединены с шаблоном родительского маршрута, и таким образом будет сформирован общий шаблон, которому должен соответствовать запрос.

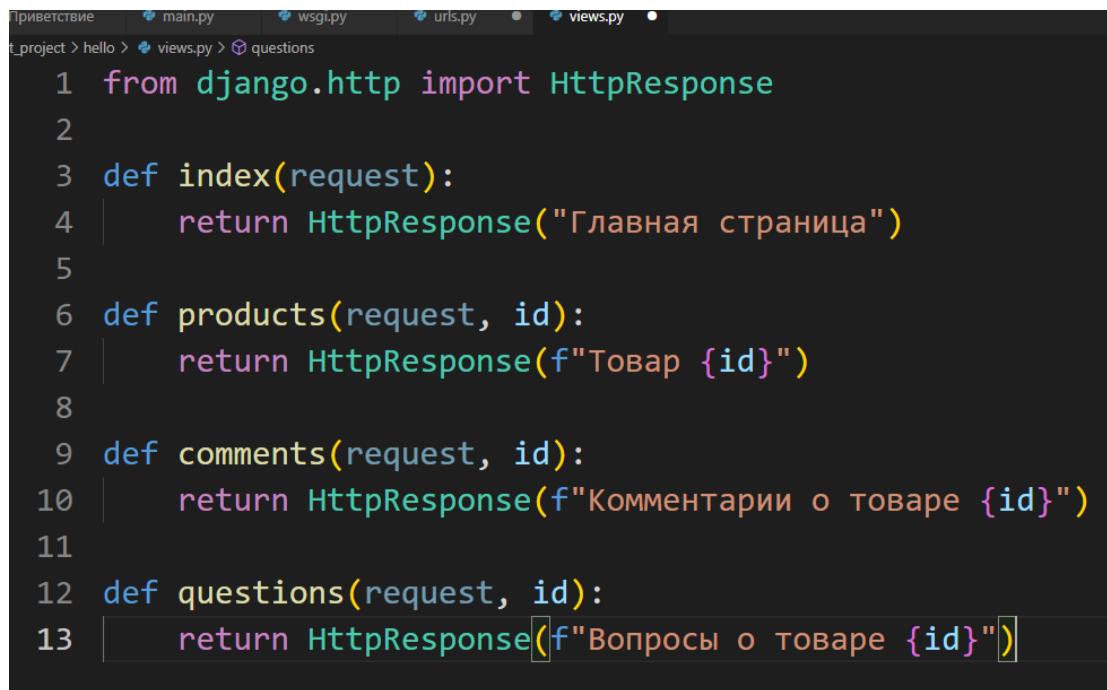
Например, если придет запрос "`http://127.0.0.1:8000/products/top`", то он будет обрабатываться функцией `top`.



Таким образом, мы можем сгруппировать маршруты для запросов, которые начинаются с определенного шаблона. Что также позволяет избежать повтора частей шаблона, когда шаблон `url` начинается с одного и того же сегмента.

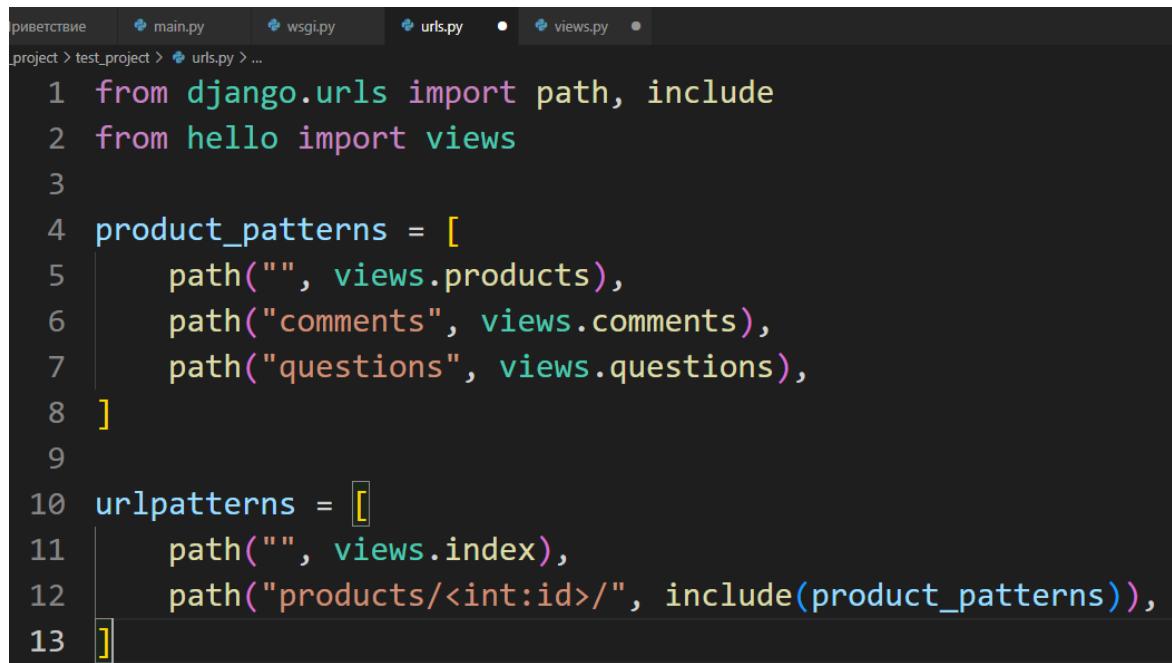
## Получение параметров

Вложенные маршруты получают параметры, определенные в родительских маршрутах. Например, определим в файле `views.py` следующие функции-представления:



```
Приветствие main.py wsgi.py urls.py views.py
t_project > hello > views.py > questions
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("Главная страница")
5
6 def products(request, id):
7     return HttpResponse(f"Товар {id}")
8
9 def comments(request, id):
10    return HttpResponse(f"Комментарии о товаре {id}")
11
12 def questions(request, id):
13     return HttpResponse(f"Вопросы о товаре {id}")
```

Изменим файл `urls.py`:



```
Приветствие main.py wsgi.py urls.py views.py
.project > test_project > urls.py > ...
1 from django.urls import path, include
2 from hello import views
3
4 product_patterns = [
5     path("", views.products),
6     path("comments", views.comments),
7     path("questions", views.questions),
8 ]
9
10 urlpatterns = [
11     path("", views.index),
12     path("products/<int:id>/", include(product_patterns)),
13 ]
```

Здесь для второго маршрута определяется числовой параметр `id`. Этот параметр передается всем вложенными маршрутам, соответственно нам не надо определять данный параметр во вложенных маршрутах

The image displays three separate browser tabs, each showing a different aspect of a product identified by the ID 10. All tabs have a similar header with icons for user profile, file, and refresh, and show the URL 127.0.0.1:8000.

- Top Tab:** Shows the product detail page for item 10. The title is "Товар 10".
- Middle Tab:** Shows the comments section for item 10. The title is "Комментарии о товаре 10".
- Bottom Tab:** Shows the questions section for item 10. The title is "Вопросы о товаре 10".

## Параметры строки запроса

От параметров, которые передаются через адрес URL, следует отличать параметры, которые передаются через строку запроса. Например, в запросе

```
http://127.0.0.1:8000/index/3/Tom/
```

два последних сегмента - 3/Tom/ представляют параметры URL или параметры маршрута. А в запросе

```
http://127.0.0.1:8000/index?id=3&name=Tom
```

те же самые значения 3 и Tom представляют **параметры строки запроса**.

Параметры строки запроса указывается после вопросительного знака ?. Каждый параметр представляет пару ключ-значение, например, в id=3 : id - название или ключ параметра, а 3 - его значение. Параметры в строке запроса отделяются друг от друга знаком амперсанда.

Для получения параметров из строки запроса применяется метод `request.GET.get()`, в которую передается название параметра.

Например, определим в файле `views.py` следующие функции:

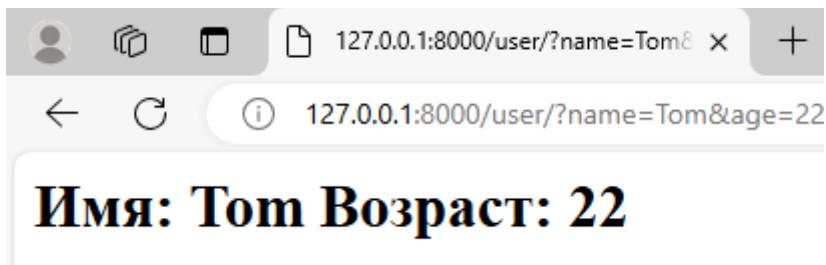
```
риветствие main.py wsgi.py urls.py views.py ×
project > hello > views.py > user
1 from django.http import HttpResponse
2
3 def index(request):
4     return HttpResponse("<h2>Главная</h2>")
5
6 def user(request):
7     age = request.GET.get("age")
8     name = request.GET.get("name")
9     return HttpResponse(f"<h2>Имя: {name} Возраст: {age}</h2>")
```

Функция `user` принимает извлекает из строки запроса два параметра: `name` и `age`.

В файле `urls.py` определим следующие маршруты:

```
риветствие main.py wsgi.py urls.py views.py ×
project > test_project > urls.py > ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6     path("user/", views.user)
7 ]
```

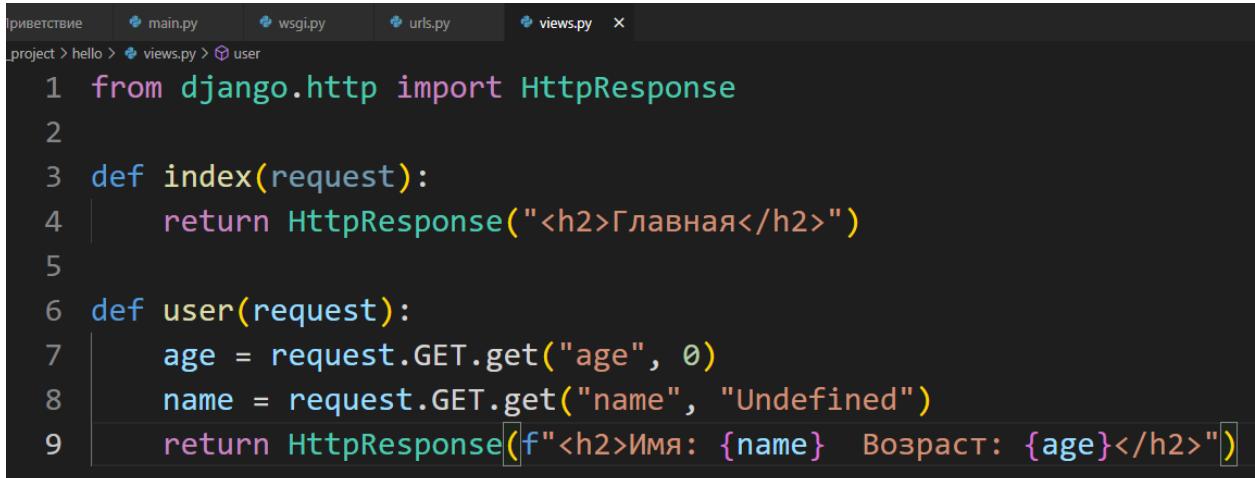
При обращении к приложению по адресу `http://127.0.0.1:8000/user/?name=Tom&age=22`, параметр name будет иметь значение "Том", а параметр age - 22.



127.0.0.1:8000/user/?name=Tom&age=22

## Имя: Том Возраст: 22

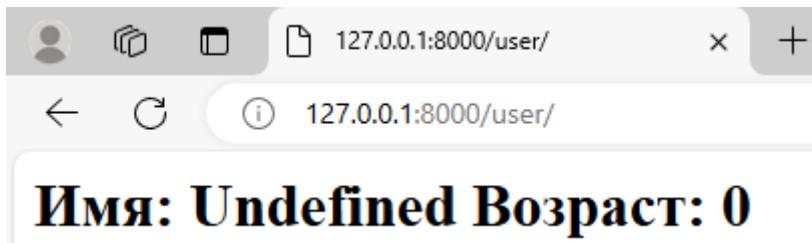
Однако возможна ситуация, когда для каких-то параметров не будет передано значение. В этом случае мы можем указать для подобных параметров значение по умолчанию. Для этого изменим определение функции user в `views.py`:



```
1 from django.http import HttpResponseRedirect
2
3 def index(request):
4     return HttpResponseRedirect("<h2>Главная</h2>")
5
6 def user(request):
7     age = request.GET.get("age", 0)
8     name = request.GET.get("name", "Undefined")
9     return HttpResponseRedirect(f"<h2>Имя: {name} Возраст: {age}</h2>")
```

Второй параметр функции `request.GET.get()` представляет значение по умолчанию для параметра, для которого не задано значение.

Соответственно при обращении по адресу `http://127.0.0.1:8000/user/` параметры name и age получат значения по умолчанию:



127.0.0.1:8000/user/

## Имя: Undefined Возраст: 0

## Переадресация и отправка статусных кодов

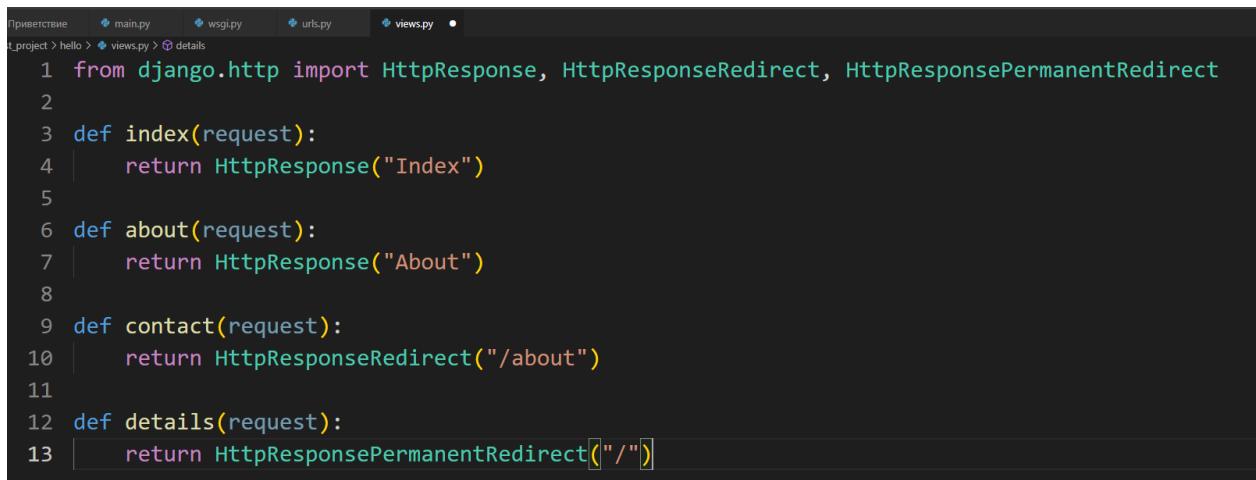
### Переадресация

При перемещении документа с одного адреса на другой мы можем воспользоваться механизмом переадресации, чтобы указать пользователям и поисковику, что документу теперь доступен по новому адресу.

Переадресация бывает временная и постоянная. При временной переадресации мы указываем, что документ временно перемещен на новый адрес. В этом случае в ответ отправляется статусный код 302. При постоянной переадресации мы уведомляем, что документ теперь постоянно будет доступен по новому адресу

Для создания временной переадресации применяется класс **HttpResponseRedirect**, а для постоянной - класс **HttpResponsePermanentRedirect**, которые расположены в пакете django.http.

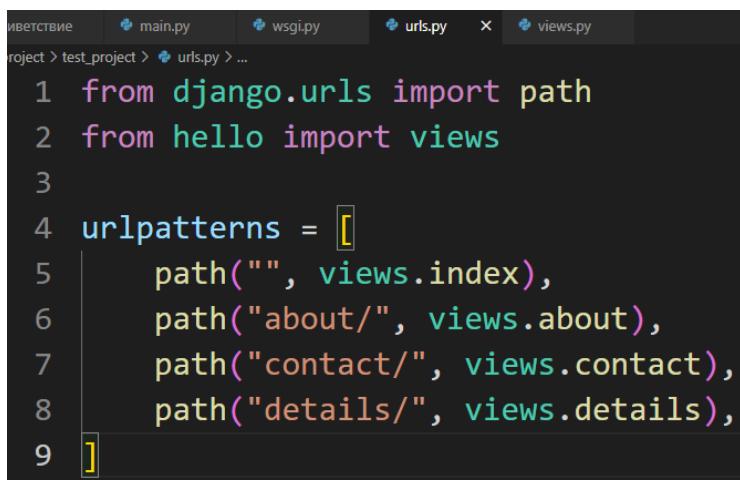
Так, определим в файле **views.py** следующий код:



```
Приветствие main.py wsgi.py urls.py views.py •
t.project > hello > views.py > details
1 from django.http import HttpResponse, HttpResponseRedirect, HttpResponsePermanentRedirect
2
3 def index(request):
4     return HttpResponse("Index")
5
6 def about(request):
7     return HttpResponse("About")
8
9 def contact(request):
10    return HttpResponseRedirect("/about")
11
12 def details(request):
13    return HttpResponseRedirectPermanentRedirect("/")
```

При обращении к функции contact она будет перенаправлять по пути "about", который будет обрабатываться функцией about. А функция details будет использовать постоянную переадресацию и перенаправлять на корень веб-приложения.

И также в файле **urls.py** для тестирования определим следующие маршруты:



```
Приветствие main.py wsgi.py urls.py views.py •
project > test_project > urls.py > ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6     path("about/", views.about),
7     path("contact/", views.contact),
8     path("details/", views.details),
9 ]
```

## Отправка статусных кодов

Также в пакете **django.http** есть ряд классов, которые позволяют отправлять определенный статусный код:

Статусный код	Класс
304 (Not Modified)	HttpResponseNotModified
400 (Bad Request)	HttpResponseBadRequest
403 (Forbidden)	HttpResponseForbidden
404 (Not Found)	HttpResponseNotFound
405 (Method Not Allowed)	HttpResponseNotAllowed
410 (Gone)	HttpResponseGone
500 (Internal Server Error)	HttpResponseServerError

В функцию конструктора этих классов можно передать некоторые данные, например, сообщение об ошибке, которое увидит пользователь:

```
HttpResponseNotModified()  
HttpResponseBadRequest("Bad Request")  
HttpResponseForbidden("Forbidden")  
HttpResponseNotFound("Not Found")  
HttpResponseNotAllowed("Method is not allowed")  
HttpResponseGone("Content is no longer here")  
HttpResponseServerError("Server Error")
```

Например, определим следующий файл **views.py**:

```
весь проект main.py wsgi.py urls.py views.py
пакет hello > views.py > index
1 from django.http import HttpResponseRedirect, HttpResponseNotFound
2 from django.http import HttpResponseRedirect, HttpResponseForbidden, HttpResponseRedirect
3
4 def index(request, id):
5     people = ["Tom", "Bob", "Sam"]
6     # если пользователь найден, возвращаем его
7     if id in range(0, len(people)):
8         return HttpResponseRedirect(people[id])
9     # если нет, то возвращаем ошибку 404
10    else:
11        return HttpResponseRedirectNotFound("Not Found")
12
13 def access(request, age):
14     # если возраст НЕ входит в диапазон 1-110, посылаем ошибку 400
15     if age not in range(1, 111):
16         return HttpResponseRedirectBadRequest("Некорректные данные")
17     # если возраст больше 17, то доступ разрешен
18     if age > 17:
19         return HttpResponseRedirect("Доступ разрешен")
20     # если нет, то возвращаем ошибку 403
21     else:
22         return HttpResponseRedirectForbidden("Доступ заблокирован: недостаточно лет")
```

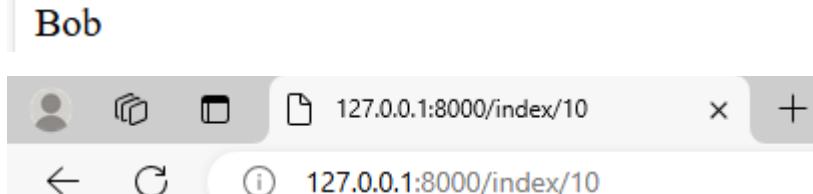
Функция `index` принимает параметр `id`. Это будет индекс элемента в списке `people`. Однако пользователь может передать и недействительный индекс, например, 100, хотя в примере в списке только 3 элемента. Поэтому, если передан действительный индекс, то возвращаем элемент по этому индексу. Если же индекс недействителен, то с помощью класса `HttpResponseNotFound` возвращаем ошибку 404 и сообщение об ошибке.

Аналогично функция `access` принимает параметр `age`, который представляет условный возраст пользователя. Если возраст выходит за некоторые разумные пределы (1-110), то с помощью класса `HttpResponseBadRequest` возвращаем ошибку 400. Если возраст укладывается в эти рамки, но он меньше 18, то с помощью класса `HttpResponseForbidden` возвращаем ошибку 403 о том, что доступ запрещен.

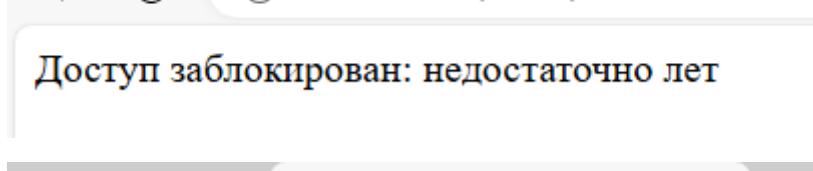
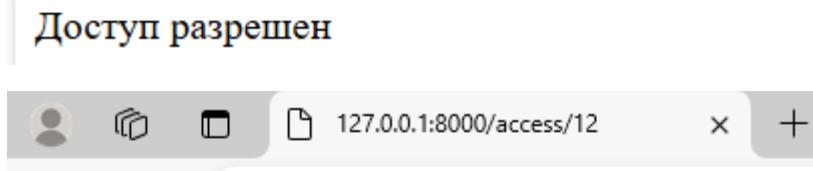
Для теста в файле **urls.py** определим следующие маршруты:

```
Приветствие main.py wsgi.py urls.py views.py
тестовый проект test_project urls.py ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("index/<int:id>", views.index),
6     path("access/<int:age>", views.access),
7 ]
```

Обратися к функции index, передав для параметра id корректные и некорректные значения. И в зависимости от значения параметра мы увидим либо данные из списка people, либо ошибку 404:



Аналогично при обращении по адресу "/access" в зависимости от значения параметра age мы увидим либо сообщения об ошибке, либо обычное сообщение:



Стоит отметить, что статусные сообщения об ошибках также отображаются в консоль запущенного приложения:

```
[07/Nov/2024 16:17:49] "GET /access/100 HTTP/1.1" 200 29
[07/Nov/2024 16:18:02] "GET /access/25 HTTP/1.1" 200 29
Forbidden: /access/12
[07/Nov/2024 16:18:15] "GET /access/12 HTTP/1.1" 403 70
Bad Request: /access/1230
[07/Nov/2024 16:18:40] "GET /access/1230 HTTP/1.1" 400 37
```

## Отправка json

За отправку клиенту данных в формате JSON в Django отвечает специальный класс - JsonResponse, который по сути представляет подкласс HttpResponseRedirect. Основная особенность JsonResponse состоит в том, что при отправке данных он автоматически устанавливает для заголовка Content-Type (тип содержимого) значение application/json

Его конструктор принимает ряд параметров:

```
def __init__(data, encoder=DjangoJSONEncoder, safe=True, json_dumps_params=None, **kwargs)
```

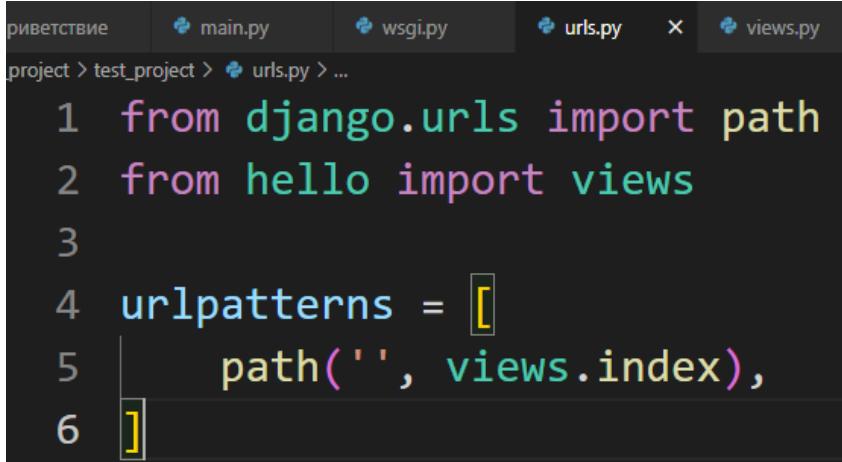
- **data**: отправляемые данные
- **encoder**: сериализатор, которые сериализует отправляемые данные в формат JSON. По умолчанию представляет тип django.core.serializers.json.DjangoJSONEncoder
- **safe**: представляет булевое значение. Если равно False, то сериализации подлежит любой объект. Если же равно True, то отправляемые данные должны представлять тип dict - то есть словарь. По умолчанию равно True
- **json\_dumps\_params**: словарь аргументов, который передается в функцию json.dumps() для генерации ответа

Например, отправим какие-нибудь данные в формате JSON. Для этого определим в файле **views.py** следующий код:



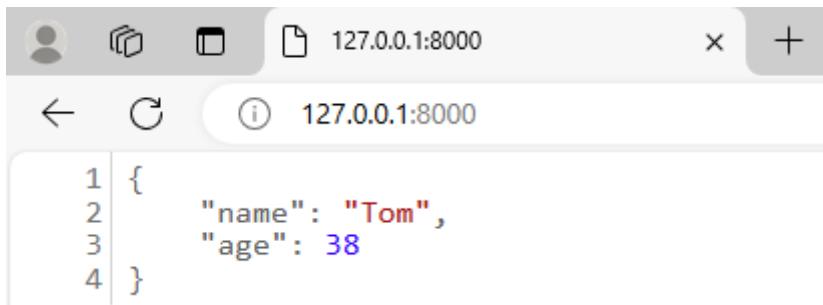
```
1 from django.http import JsonResponse
2
3 def index(request):
4     return JsonResponse({"name": "Tom", "age": 38})
5
```

Поэтому изменим файл **urls.py** следующим образом:



```
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path('', views.index),
6 ]
```

В данном случае отправляется словарь с двумя элементами name и age. И при обращении в браузере к функции index мы увидим эти данные:



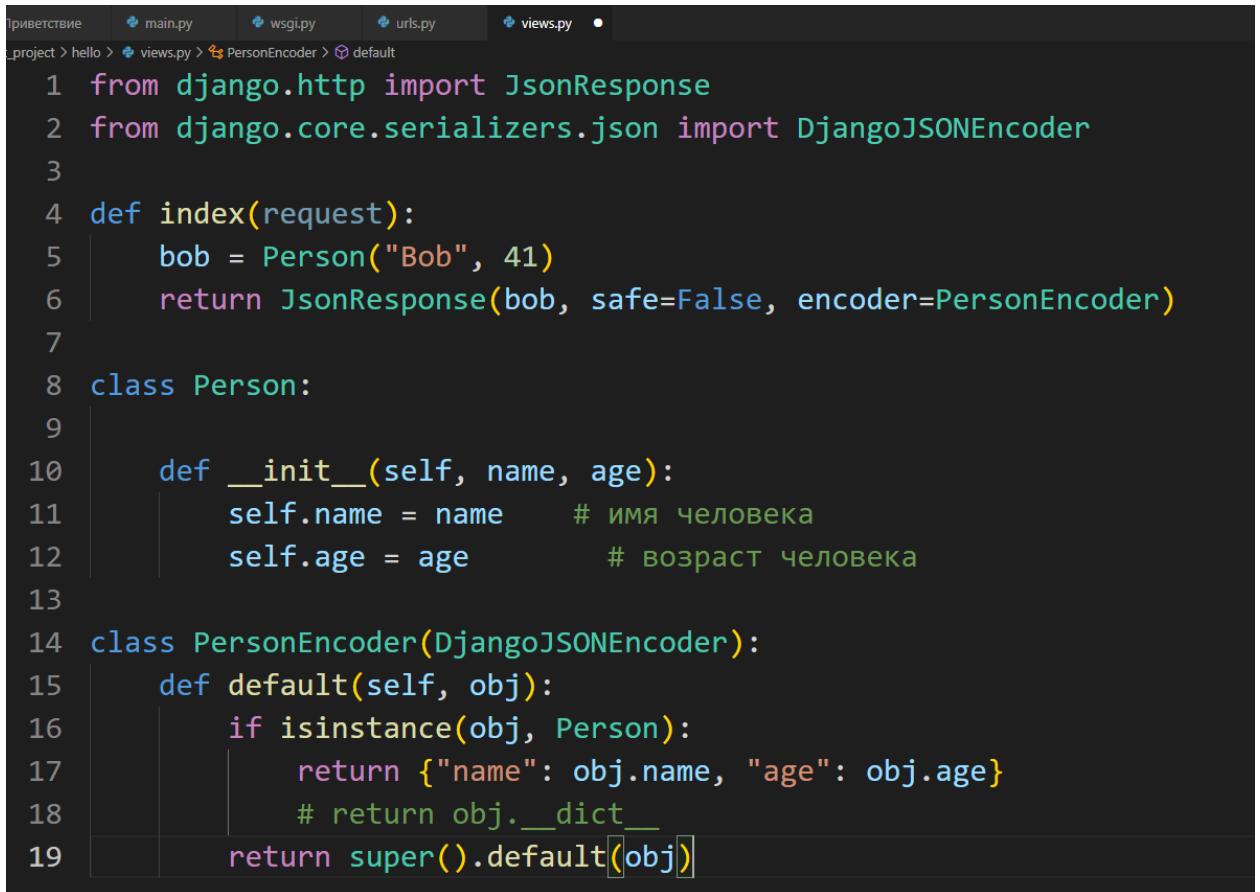
A screenshot of a web browser window. The address bar shows '127.0.0.1:8000'. The page content displays the following JSON object:

```
1 {
2     "name": "Tom",
3     "age": 38
4 }
```

## Сериализация произвольных объектов

По умолчанию JsonResponse сериализует и отправляет только словари. Однако что, если мы хотим отправить объект какого-то своего типа? В этом случае необходимо определить класс-сериализатор, который будет содержать логику сериализации объекта в json.

Например:



A screenshot of a code editor showing the contents of the `views.py` file. The code defines a `Person` class and a `PersonEncoder` class that extends `DjangoJSONEncoder`. The `index` view returns a `JsonResponse` object containing an instance of `Person`, with `safe=False` and `encoder=PersonEncoder`.

```
1 from django.http import JsonResponse
2 from django.core.serializers.json import DjangoJSONEncoder
3
4 def index(request):
5     bob = Person("Bob", 41)
6     return JsonResponse(bob, safe=False, encoder=PersonEncoder)
7
8 class Person:
9
10    def __init__(self, name, age):
11        self.name = name      # имя человека
12        self.age = age        # возраст человека
13
14 class PersonEncoder(DjangoJSONEncoder):
15    def default(self, obj):
16        if isinstance(obj, Person):
17            return {"name": obj.name, "age": obj.age}
18            # return obj.__dict__
19        return super().default(obj)
```

В данном случае JsonResponse отправляет объект типа Person, у которого определены два атрибута: name и age.

Объект Person не является словарем, поэтому параметр `safe` имеет значение `False`. Кроме того, параметр `encoder` указывает на сериализатор, который будет сериализовать данные в json. В данном случае это класс `PersonEncoder`.

Класс сериализатора наследуется от `django.core.serializers.json.DjangoJSONEncoder`. Он реализует метод `default`, который возвращает сериализованный объект. В частности, в этом методе сначала проверяем, представляет ли параметр объект `Person`. И если представляет, то возвращаем словарь из значений атрибутов объекта

```
return {"name": obj.name, "age": obj.age}
```

Стоит отметить, что в данном случае мы можем просто возвратить представление объекта в виде словаря

```
return {"name": obj.name, "age": obj.age}
```

Однако в отдельных ситуациях может потребоваться более тонкая настройка сериализации.

Если же объект не представляет тип `Person`, то передаем его в реализацию метода `default` родительского класса.

Результат при обращении в браузере:



A screenshot of a web browser window. The address bar shows '127.0.0.1:8000'. The page content displays the following JSON code:

```
1 {  
2     "name": "Bob",  
3     "age": 41  
4 }
```

## Отправка и получение кук

Куки представляют самый простой способ сохранить данные пользователя. Куки хранятся на компьютере пользователя и могут устанавливаться как на сервере, так и на клиенте. Так как куки посылаются с каждым запросом на сервер, то их максимальный размер ограничен 4096 байтами. Рассмотрим, как отправить клиенту и получить от клиента куки в приложении на Django.

### Установка куки

За установку куки и отправку их клиенту отвечает метод `set_cookie()` класса `HttpResponse`. Этот метод имеет следующую сигнатуру:

```
set_cookie(key, value='', max_age=None, expires=None, path='/', domain=None, secure=False,  
httponly=False, samesite=None)
```

Параметры метода:

- **key**: ключ или имя куки
- **value**: значение куки
- **max\_age**: максимальное время жизни куки в секундах. Это может быть либо объект `timedelta`, либо число, либо значение `None` (ограничивает время жизни куки текущей сессией браузера, является значением по умолчанию).
- **expires**: время и дата, когда истекает действие куки. Должен представлять строку в формате "Wdy, DD-Mon-YY HH:MM:SS GMT" или объект `datetime.datetime`
- **path**: путь, для которого устанавливаются куки
- **domain**: домен, к которому применяются куки
- **secure**: устанавливает используемый протокол. Так, если имеет значение `True`, то куки будут посыпаться на сервер только в запросе по протоколу `https`
- **httponly**: устанавливает доступность для скриптов javascript на клиенте. Так, значение `httponly=True` предотвращает доступ к куки из кода javascript на клиенте
- **samesite**: устанавливает разрешения на отправку куки в кроссдоменных запросах. Так, значения `samesite='Strict'` и `samesite='Lax'` указывают браузеру не посылать куки в кроссдоменных запросах. Значение по умолчанию `samesite='None'` разрешает отправку куки в кроссдоменных запросах

Также класс `HttpResponse` предоставляет еще один метод для установки кук:

```
set_signed_cookie(key, value, salt='', max_age=None, expires=None, path='/', domain=None,  
secure=False, httponly=False, samesite=None)
```

Данный метод принимает те же параметры, его отличие в том, что он применяет шифрование. Необязательный параметр `salt` позволяет задать соль для шифрования.

Например, установим куки. Пусть в файле `views.py` имеется следующая функция:

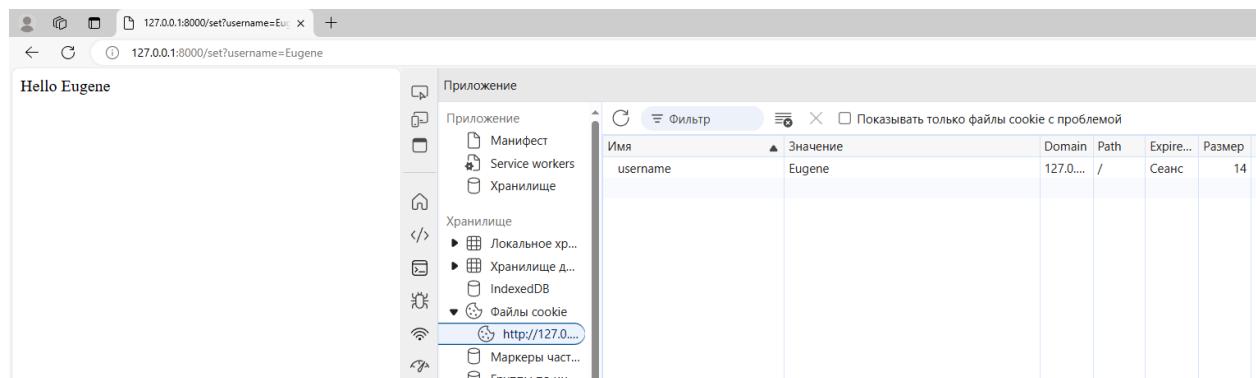
```
Приветствие main.py wsgi.py urls.py views.py ●
.project > hello > views.py > set
1 from django.http import HttpResponseRedirect
2
3 # установка куки
4 def set(request):
5     # получаем из строки запроса имя пользователя
6     username = request.GET.get("username", "Undefined")
7     # создаем объект ответа
8     response = HttpResponseRedirect(f"Hello {username}")
9     # передаем его в куки
10    response.set_cookie("username", username)
11
12    return response
```

Из строки запроса получаем значение параметра "username" и передаем его в куки по одноименному ключу

Пусть в файле `urls.py` эта функция вызывается в запросе по пути "/set":

```
Приветствие main.py wsgi.py urls.py views.py
test_project > test_project > urls.py > ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("set", views.set),
6 ]
```

Обратимся к по адресу "/set", передав через строку запроса параметр `username`, и сервер установит куки, а браузер сохранит их. И мы сможем их увидеть через инструменты разработчика:



## Получение куки

Если куки не шифрованные, то для их получения у объекта HttpRequest можно использовать атрибут **COOKIES**, который представляет словарь.

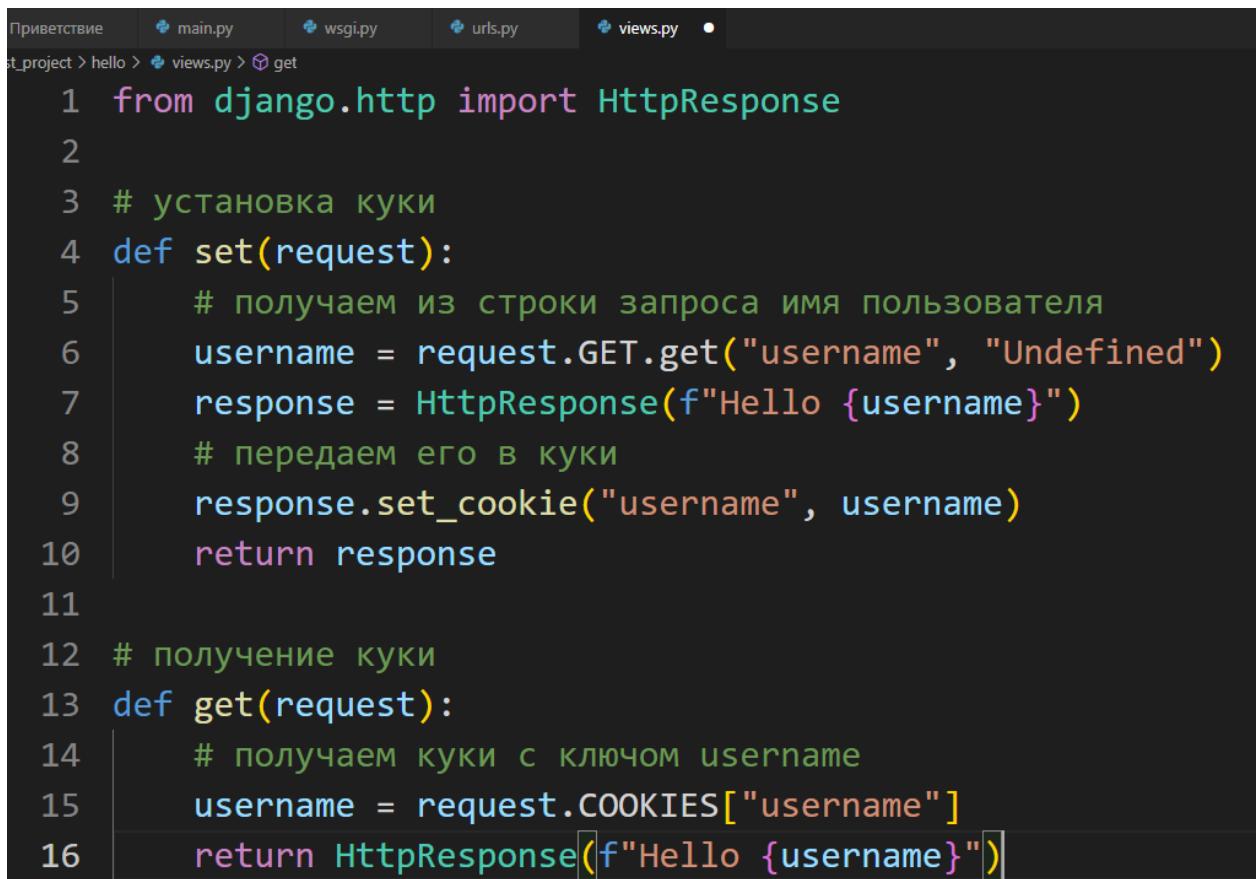
Если куки шифрованные методом set\_signed\_cookie, то для их получения в классе HttpRequest применяется метод

```
get_signed_cookie(key, default=RAISE_ERROR, salt='', max_age=None)
```

Параметры метода:

- **key**: ключ куки, которые надо получить
- **default**: значение по умолчанию, если куки с указанным ключом отсутствуют в запросе
- **max\_age**: максимальное время жизни куки в секундах
- **salt**: соль шифрования, должен иметь то же самое значение, которое передавалось в set\_signed\_cookie

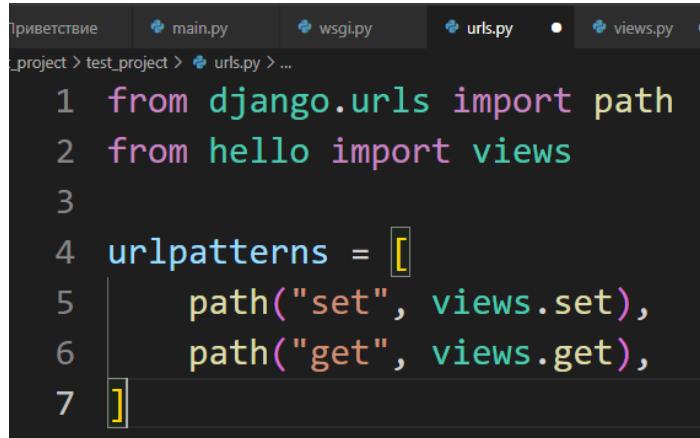
Например, получим ранее установленные куки по ключу. Определим в файле **views.py** дополнительную функцию **get()**:



```
Приветствие main.py wsgi.py urls.py views.py
st_project > hello > views.py > get
1 from django.http import HttpResponse
2
3 # установка куки
4 def set(request):
5     # получаем из строки запроса имя пользователя
6     username = request.GET.get("username", "Undefined")
7     response = HttpResponse(f"Hello {username}")
8     # передаем его в куки
9     response.set_cookie("username", username)
10    return response
11
12 # получение куки
13 def get(request):
14     # получаем куки с ключом username
15     username = request.COOKIES["username"]
16     return HttpResponse(f"Hello {username}")
```

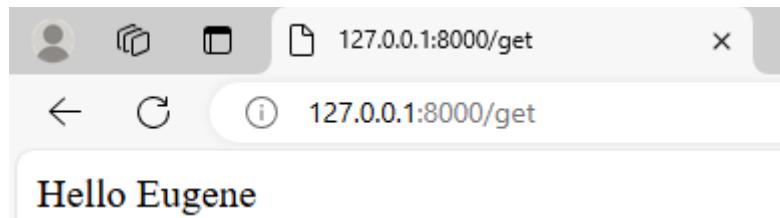
С помощью выражения `request.COOKIES["username"]` получаем куки по ключу "username" и передаем их значение в ответ клиенту.

Пусть в файле **urls.py** эта функция вызывается в запросе по пути "/get":



```
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("set", views.set),
6     path("get", views.get),
7 ]
```

Обратимся к по адресу "/get" и получим ранее установленную куку username:

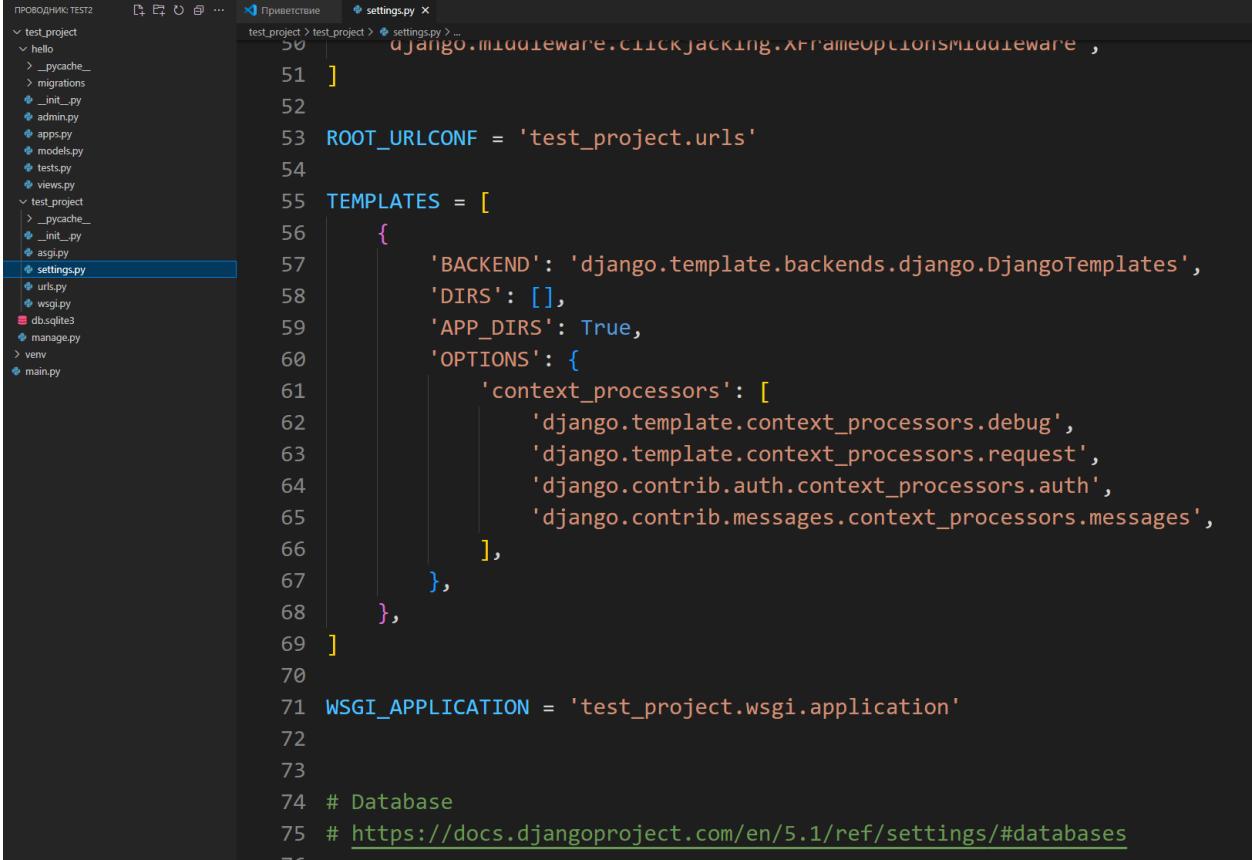


## Шаблоны

### Создание и использование шаблонов

Шаблоны (templates) отвечают за формирование внешнего вида приложения. Они предоставляют специальный синтаксис, который позволяет внедрять данные в код HTML.

Допустим, у нас есть проект **test\_project**, и в нем определено одно приложение - **hello**:



The screenshot shows the PyCharm IDE interface. On the left, the project tree displays a 'test\_project' folder containing 'hello' (with 'migrations' and '\_pycache\_'), 'migrations', 'admin.py', 'apps.py', 'models.py', 'tests.py', and 'views.py'. Below these are 'test\_project' (with '\_pycache\_'), 'settings.py' (selected), 'urls.py', 'wsgi.py', 'db.sqlite3', and 'manage.py'. A 'venv' folder is also present. On the right, the code editor shows the 'settings.py' file with the following content:

```
50 ]
51 ]
52
53 ROOT_URLCONF = 'test_project.urls'
54
55 TEMPLATES = [
56     {
57         'BACKEND': 'django.template.backends.django.DjangoTemplates',
58         'DIRS': [],
59         'APP_DIRS': True,
60         'OPTIONS': {
61             'context_processors': [
62                 'django.template.context_processors.debug',
63                 'django.template.context_processors.request',
64                 'django.contrib.auth.context_processors.auth',
65                 'django.contrib.messages.context_processors.messages',
66             ],
67         },
68     },
69 ]
70 WSGI_APPLICATION = 'test_project.wsgi.application'
71
72
73
74 # Database
75 # https://docs.djangoproject.com/en/5.1/ref/settings/#databases
76
```

Настройка функциональности шаблонов в проекте Django производится в файле **settings.py**. с помощью переменной **TEMPLATES**.

Данная переменная принимает список конфигураций для каждого движка шаблонов. По умолчанию определена одна конфигурация, которая имеет следующие параметры

- **BACKEND**: движок шаблонов. По умолчанию применяется встроенный движок **django.template.backends.django.DjangoTemplates**
- **DIRS**: определяет список каталогов, где движок шаблонов будет искать файлы шаблонов. По умолчанию пустой список
- **APP\_DIRS**: указывает, будет ли движок шаблонов искать шаблоны внутри папок приложений в папке templates.
- **OPTIONS**: определяет дополнительный список параметров

Итак, в конфигурации по умолчанию параметр **APP\_DIRS** имеет значение **True**, а это значит, что движок шаблонов будет также искать нужные файлы шаблонов в папке

приложения в каталоге **templates**. То есть по умолчанию мы уже имеем настроенную конфигурацию, готовую к использованию шаблонов. Теперь определим сами шаблоны.

Добавим в папку приложения каталог **templates**. А в нем определим файл **index.html** и определим следующий код:

The screenshot shows a code editor with the following file structure on the left:

```
ПРОВОДНИК: TEST2
└── test_project
    ├── hello
    │   ├── __pycache__
    │   ├── migrations
    │   └── templates
    │       └── index.html
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── models.py
    ├── tests.py
    ├── views.py
    └── test_project
        ├── __pycache__
        ├── __init__.py
        └── wsgi.py
```

The right pane displays the content of `index.html`:

```
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <h2>Hello World</h2>
9 </body>
10 </html>
```

По сути, это обычная веб-страница, которая содержит код html. Теперь используем эту страницу для отправки ответа пользователю. И для этого перейдем в приложении `hello` к файлу **views.py**, который определяет функции для обработки запроса. Изменим этот файл следующим образом:

The screenshot shows the content of `views.py`:

```
1 from django.shortcuts import render
2
3 def index(request):
4     return render(request, "index.html")
```

Из модуля `django.shortcuts` импортируется функция **render**.

Функция `index` вызывает функцию `render`, которой передаются объект запроса `request` и путь к файлу шаблона в рамках папки `templates` - "`index.html`".

В файле **urls.py** проекта пропишем сопоставление функции `index` с запросом к корню веб-приложения:

The screenshot shows the content of `urls.py`:

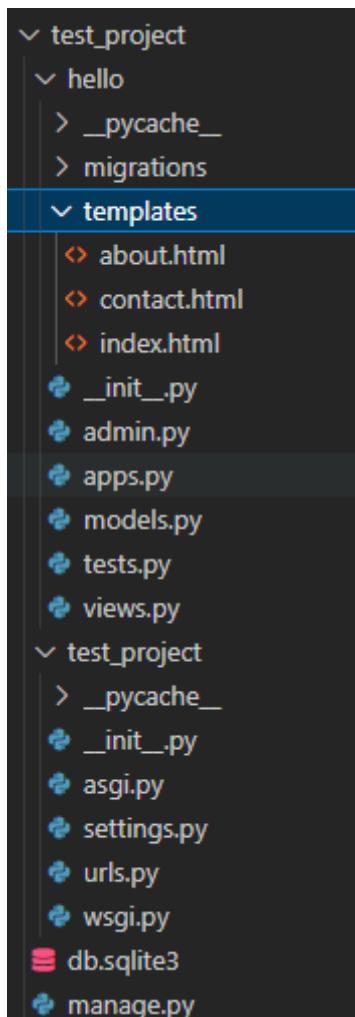
```
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6 ]
```

И запустим проект на выполнение и перейдем к приложению в браузере (если проект запущен, то его надо перезапустить):

```
PS C:\Users\Vital\Python\Test2> venv\scripts\activate  
(venv) PS C:\Users\Vital\Python\Test2> cd test_project  
(venv) PS C:\Users\Vital\Python\Test2\test_project> python manage.py runserver
```



Подобным образом можно указать и другие шаблоны. Например, в папку **templates** добавим еще две страницы: **about.html** и **contact.html**



И также в файле **views.py** определим функции, которые используют данные шаблоны:

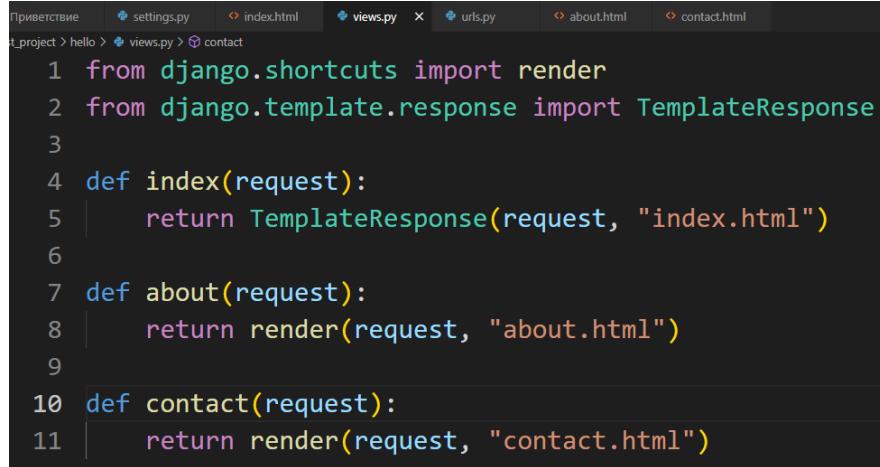
```
Приветствие settings.py index.html views.py urls.py about.html contact.html  
т.р. проект > hello > views.py > contact  
1 from django.shortcuts import render  
2  
3 def index(request):  
4     return render(request, "index.html")  
5  
6 def about(request):  
7     return render(request, "about.html")  
8  
9 def contact(request):  
10    return render(request, "contact.html")
```

А в файле **urls.py** свяжем функции с маршрутами:

```
Приветствие settings.py index.html views.py urls.py about.html contact.html  
т.р. проект > test_project > urls.py > ...  
1 from django.urls import path  
2 from hello import views  
3  
4 urlpatterns = [  
5     path("", views.index),  
6     path("about/", views.about),  
7     path("contact/", views.contact),  
8 ]
```

## TemplateResponse

Выше для генерации шаблона применялась функция `render()`, которая является наиболее распространенным вариантом. Однако также мы можем использовать класс `TemplateResponse`:



```
Приветствие settings.py index.html views.py urls.py about.html contact.html
t_project > hello > views.py > contact
  1 from django.shortcuts import render
  2 from django.template.response import TemplateResponse
  3
  4 def index(request):
  5     return TemplateResponse(request, "index.html")
  6
  7 def about(request):
  8     return render(request, "about.html")
  9
 10 def contact(request):
 11     return render(request, "contact.html")
```

Результат будет тот же самый.

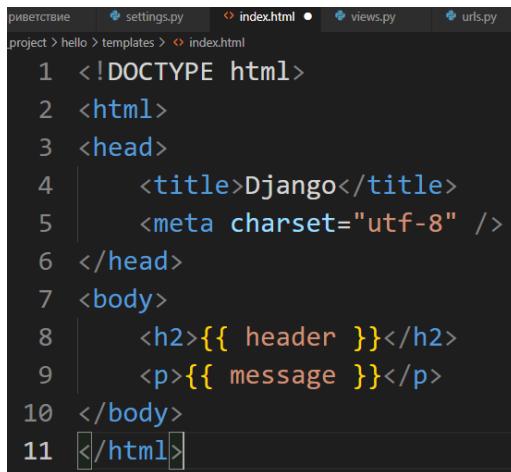
## Передача данных в шаблоны

Одним из преимуществ шаблонов является то, что мы можем передать в них динамически из представлений различные данные. Для вывода данных в шаблоне могут использоваться различные способы. Для вывода самых простых данных применяется двойная пара фигурных скобок:

```
 {{ название_объекта }}
```

Например, пусть в проекте у нас есть папка templates, в которой содержится шаблон **index.html**.

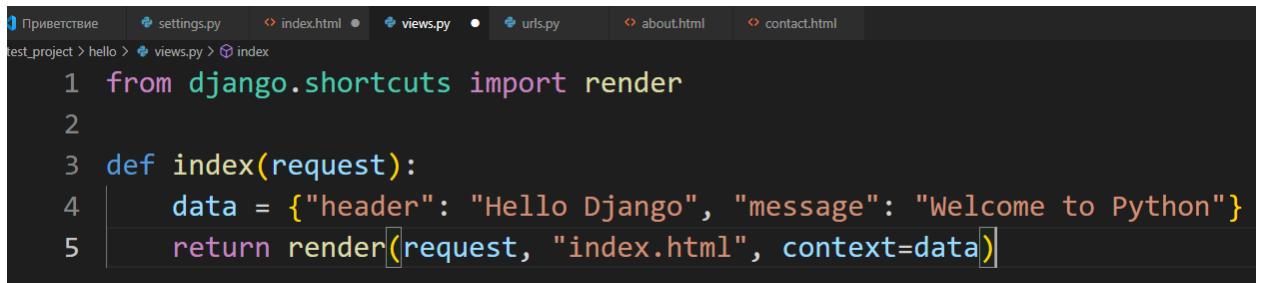
Определим в файле **index.html** следующий код:



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <h2>{{ header }}</h2>
9     <p>{{ message }}</p>
10 </body>
11 </html>
```

Здесь используется две переменных: `message` и `header`. Они будут передаваться из представления.

Чтобы из функции-представления передать данные в шаблон применяется третий параметр функции `render`, который еще называется `context` и который представляет словарь. Например, изменим файл **views.py** следующим образом:



```
1 from django.shortcuts import render
2
3 def index(request):
4     data = {"header": "Hello Django", "message": "Welcome to Python"}
5     return render(request, "index.html", context=data)
```

В шаблоне используются две переменных, соответственно словарь, который передается в функцию `render` через параметр `context`, теперь содержит два значения с ключами `header` и `message`.

А в файле **urls.py** свяжем функцию с маршрутом:

```
внтижение settings.py index.html views.py urls.py
ект > test_project > urls.py > ...
1 from django.urls import path
2 from hello import views
3
4 urlpatterns = [
5     path("", views.index),
6 ]
```

В результате при обращении к корню веб-приложения мы увидим следующий вывод в браузере:



## Передача сложных данных

Рассмотрим передачу более сложных данных. Допустим, в представлении передаются следующие данные:

```
риветствие settings.py index.html views.py urls.py about.html contact.html
project > hello > views.py ...
1 from django.shortcuts import render
2
3 def index(request):
4     header = "Данные пользователя"                      # обычная переменная
5     langs = ["Python", "Java", "C#"]                   # список
6     user = {"name": "Том", "age": 23}                  # словарь
7     address = ("Абрикосовая", 23, 45)                # кортеж
8
9     data = {"header": header, "langs": langs, "user": user, "address": address}
10    return render(request, "index.html", context=data)
```

В качестве третьего параметра в функцию `render` нам надо передать словарь, поэтому все данные обрабатываются в словарь и в таком виде передаются в шаблон.

В этом случае шаблон мог бы выглядеть, например, следующим образом:

```
риветствие settings.py index.html views.py urls.py about.html contact.html
project > hello > templates > index.html
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <h1>{{ header }}</h1>
9     <p>Имя: {{ user.name }} Возраст: {{ user.age }}</p>
10    <p>Адресс: ул. {{ address.0 }}, д. {{ address.1 }}, кв. {{ address.2 }}</p>
11    <p>Языки: {{ langs.0 }}, {{ langs.1 }}</p>
12 </body>
13 </html>
```

Поскольку объекты `langs` и `address` представляют соответственно массив и кортеж, то мы можем обратиться к их элементам через индексы, как мы бы работали бы с ними в коде на Python, например, первый элемент кортежа `address`: `address.0`.

Подобным образом, поскольку объект `user` представляет словарь, то мы можем обратиться к его элементам по ключам `name` и `age`: `{{ user.name }}` `{{ user.age }}`.

В итоге мы получим следующий вывод в веб-браузере:

Данные пользователя

Имя: Tom Возраст: 23

Адресс: ул. Абрикосовая, д. 23, кв. 45

Языки: Python, Java

## TemplateResponse

Если для генерации шаблона применяется класс **TemplateResponse**, то в его конструктор также через третий параметр можно передать данные для шаблона:

```
1 from django.template.response import TemplateResponse
2
3 def index(request):
4     header = "Данные пользователя"                      # обычная переменная
5     langs = ["Python", "Java", "C#"]                   # список
6     user = {"name": "Tom", "age": 23}                  # словарь
7     address = ("Абрикосовая", 23, 45)                # кортеж
8
9     data = {"header": header, "langs": langs, "user": user, "address": address}
10    return TemplateResponse(request, "index.html", data)
```

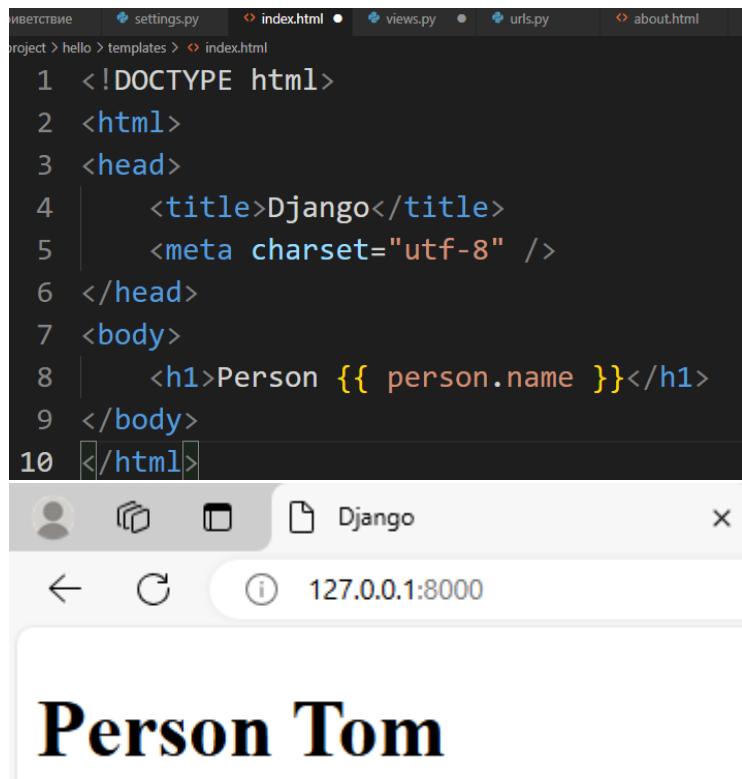
## Передача объектов классов

Подобным образом можно передавать в шаблоны объекты своих классов. Например, определение функции-представления:

```
1 from django.shortcuts import render
2
3 def index(request):
4     return render(request, "index.html", context = {"person": Person("Tom")})
5
6 class Person:
7
8     def __init__(self, name):
9         self.name = name      # имя человека
```

Здесь определяется класс Person, в конструкторе которого передается некоторое значение для атрибута name. В функции index в шаблон передается объект с ключом "person".

В шаблоне **index.html** мы можем обращаться к функциональности объекта, например, к его атрибуту `name`:



The code editor shows the `index.html` file with the following content:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <h1>Person {{ person.name }}</h1>
9 </body>
10 </html>
```

The browser window shows the rendered output: **Person Tom**.

## Встроенные теги шаблонов

Django предоставляет возможность использовать в шаблонах ряд специальных тегов, которые упрощают вывод некоторых данных. Рассмотрим некоторые наиболее используемые теги.

### autoescape

Тег **autoescape** позволяет автоматически экранировать ряд символов HTML и тем самым сделать вывод на страницу более безопасным. В частности, производятся следующие замены:

- < заменяется на &lt;
- > заменяется на &gt;
- ' (одинарная кавычка) заменяется на &#x27;
- " (двойная кавычка) заменяется на &quot;
- & заменяется на &amp;

Например, пусть у нас будет определен следующий шаблон **index.html**:

```
риветствие settings.py index.html views.py urls.py
project > hello > templates > index.html
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     {{ body }}
9
10    {% autoescape off %}
11        {{ body }}
12    {% endautoescape %}
13 </body>
14 </html>
```

Здесь в шаблоне два раза выводится содержимое переменной `body`. Однако сначала она выводится сама по себе, в во втором случае ее вывод помещается в между тегами `{% autoescape %}` и `{% endautoescape %}`. Тег в качестве параметра принимает одно из значений: `on` (экранирование включено) и `off` (экранирование отключено). В данном случае тегу передается значение `off`, а это значит, что внутри блока тега `autoescape` экранирование отключено.

Определим следующую функцию-представление, которая использует данный шаблон и передает в него некоторые данные:

```
риветствие settings.py index.html views.py urls.py about.html contact.html
project > hello > views.py > index
1 from django.shortcuts import render
2
3 def index(request):
4     return render(request, "index.html", context = {"body": "<h1>Hello World!</h1>"})
```

Здесь в шаблон передается некоторая строка с кодом HTML. По умолчанию шаблоны применяют экранирование. Поэтому на веб-странице мы увидим код `html` в текстовом виде, а во втором случае переданные данные будут интерпретированы непосредственно как код `html`:



Для определения комментариев в шаблоне применяется тег **comment**: все, что помещается между тегами `{% comment %}` и `{% endcomment %}`, игнорируется при генерации html-страницы.

```
известные settings.py index.html views.py urls.py
project > hello > templates > index.html
1 1 <!DOCTYPE html>
2 2 <html>
3 3 <head>
4 4   <title>Django</title>
5 5   <meta charset="utf-8" />
6 6 </head>
7 7 <body>
8 8
9 9   {% comment %}
10 10     <p>Вывод содержимого в шаблон Django</p>
11 11   {% endcomment %}
12 12
13 13   {{ body }}
14 14 </body>
15 15 </html>
```

### if..else

Тег **if** оформляется в виде блока:

```
{% if условие %}
    содержимое блока
{% endif %}
```

Этот тег оценивает некоторое условие, которое должно возвращать True или False: если возвращается **True**, то выводится содержимое блока `{% if %}`.

Например, пусть в представлении передаются в шаблон некоторые значения:

```
известные settings.py index.html views.py urls.py
project > hello > views.py > index
1 1 from django.shortcuts import render
2
3 3 def index(request):
4 4     data = {"n" : 5}
5 5     return render(request, "index.html", context=data)
```

В шаблоне в зависимости от значения переменной n мы можем выводить определенный контент:

```
{% if n > 0 %}
    <p>Число положительное</p>
{% endif %}
```

То есть если n больше 0, то будет выводиться, что число положительное. Если n меньше или равно 0, ничего не будет выводиться.

С помощью дополнительного тега `{% else %}` можно вывести контент в случае, если условие после if равно False:

```
{% if n > 0 %}
    <p>Число положительное</p>
{% else %}
    <p>Число отрицательное или равно нулю</p>
{% endif %}
```

С помощью тега `{% elif %}` можно проверить дополнительные условия, если условие в if равно False:

```
{% if n > 0 %}
    <p>Число положительное</p>
{% elif n < 0 %}
    <p>Число отрицательное</p>
{% else %}
    <p>Число равно нулю</p>
{% endif %}
```

Стоит отметить, что в условии, которое передается тегу if, можно применять ряд операторов, которые возвращают True или False: ==, !=, <, >, <=, >=, in, not in, is, is not, and, or, not.

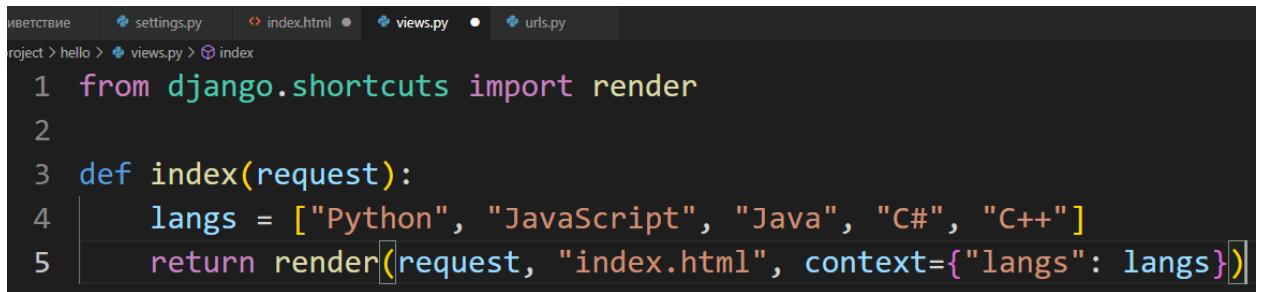
```
{% if a > 0 and b > 0 %}
    <p>{{ a }}</p>
{% else %}
    <p>{{ b }}</p>
{% endif %}
```

## Циклы

Тег `for` позволяет создавать циклы. Этот тег принимает в качестве параметра некоторую коллекцию и пробегается по этой коллекции, обрабатывая каждый ее элемент.

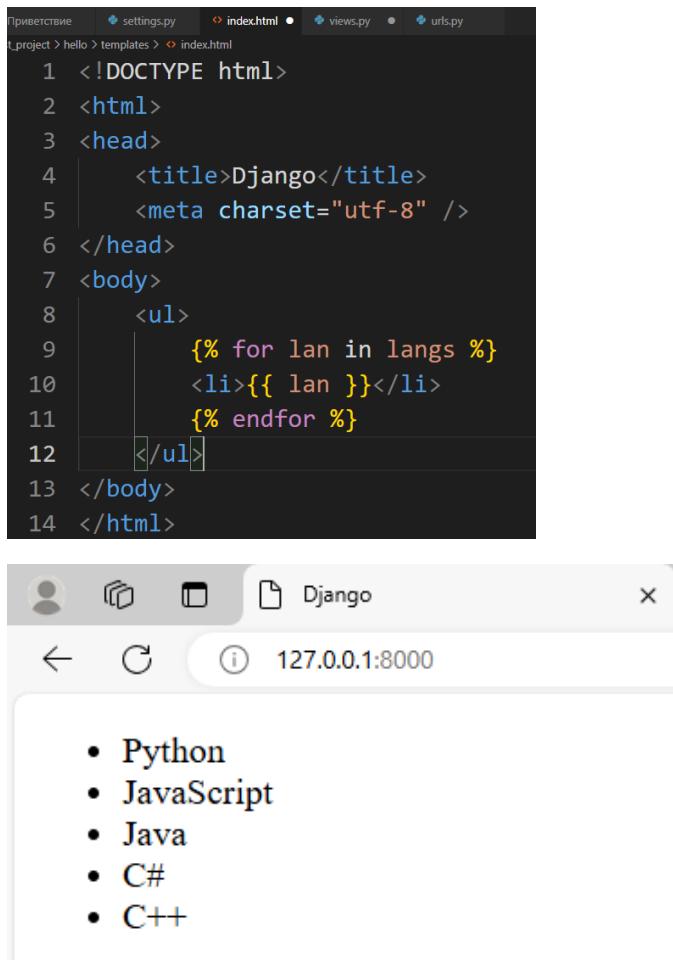
```
{% for element in collection %}
    действия с element
{% endfor %}
```

Например, пусть из представления в шаблон передается некоторый список:

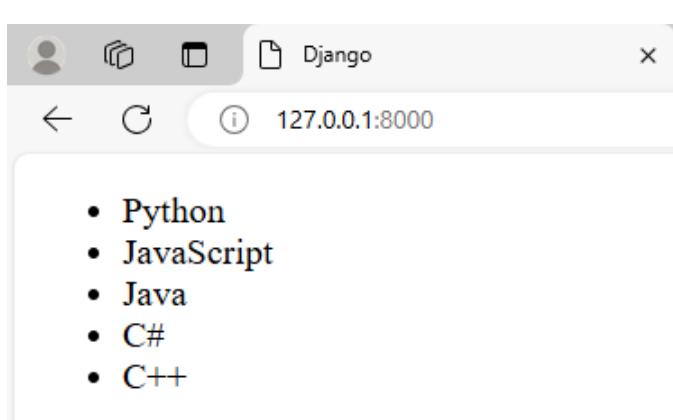


```
1 from django.shortcuts import render
2
3 def index(request):
4     langs = ["Python", "JavaScript", "Java", "C#", "C++"]
5     return render(request, "index.html", context={"langs": langs})
```

Выведем элементы списка langs в шаблоне с помощью тега **for**:



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <ul>
9         {% for lan in langs %}
10            <li>{{ lan }}</li>
11        {% endfor %}
12    </ul>
13 </body>
14 </html>
```



Django

127.0.0.1:8000

- Python
- JavaScript
- Java
- C#
- C++

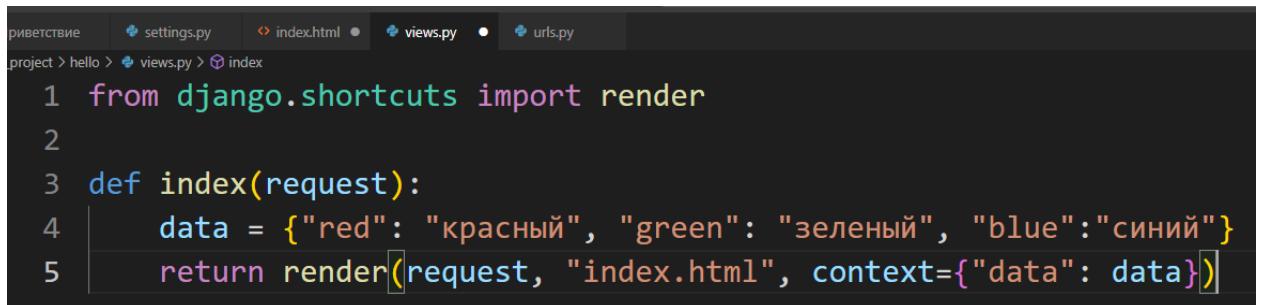
Вполне возможно, что переданная из представления в шаблон коллекция окажется пустой. На этот случай мы можем использовать тег **{% empty %}**:

```
<ul>
    {% for lan in langs %}
        <li>{{ lan }}</li>
    {% empty %}
        <li>Langs array is empty</li>
    {% endfor %}
</ul>
```

Подобным образом можно перебирать в шаблоне другие коллекции, например, словари:

```
{% for key, value in data.items %}
    <p>{{ key }}: {{ value }}</p>
{% endfor %}
```

В данном случае перебирается словарь data, который передается из представления:



```
1 from django.shortcuts import render
2
3 def index(request):
4     data = {"red": "красный", "green": "зеленый", "blue": "синий"}
5     return render(request, "index.html", context={"data": data})
```

### Определение переменных

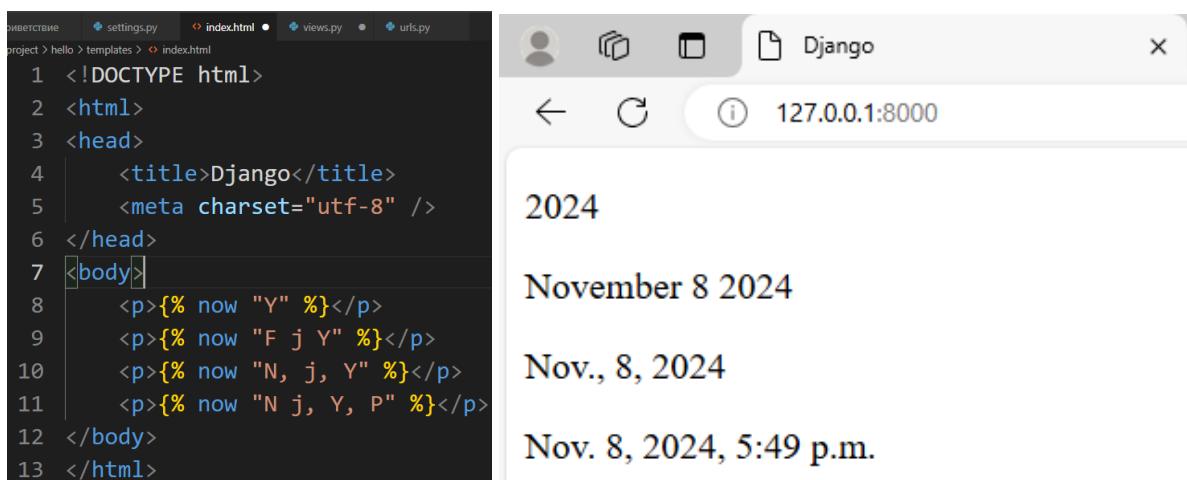
Тег `{% with %}` позволяет определить переменную и использовать ее внутри блока тега. Например:

```
{% with name="Tom" age=29 %}
    <div>
        <p>Name: {{ name }}</p>
        <p>Age: {{ age }}</p>
    </div>
{% endwith %}
```

В данном случае определены две переменных: name и age, которые мы можем использовать внутри блока `{% with %} ... {% endwith %}`. Однако вне этого блока эти переменные использоваться не могут.

### Даты

Тег `{% now "формат_данных" %}` позволяет вывести системное время. В качестве параметра тегу now передается формат даты, который указывает, как форматировать время и дату.



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <p>{{ now "Y" }}</p>
9     <p>{{ now "F j Y" }}</p>
10    <p>{{ now "N, j, Y" }}</p>
11    <p>{{ now "N j, Y, P" }}</p>
12 </body>
13 </html>
```

Django

127.0.0.1:8000

2024

November 8 2024

Nov., 8, 2024

Nov. 8, 2024, 5:49 p.m.

Символ "Y" передает год в виде четырех цифр, "y" - берет из года последние две цифры. "F" - полное название месяца, "N" -сокращенное название месяца. "j" - день месяца. "P" - время. Все возможные форматы для вывода даты и времени можно посмотреть в [документации](#)

Стоит отметить, что в качестве времени Django берет значение переменной из файла **settinds.py**, которая по умолчанию имеет следующее определение:

```
TIME_ZONE = 'UTC'
```

То есть отсчитывается относительно UTC. Это может быть не всегда удобно. И соответственно мы можем изменить ее значение на нужное. Например, настройка для московского часового пояса:

```
TIME_ZONE = 'Europe/Moscow'
```

## Фильтры шаблонов

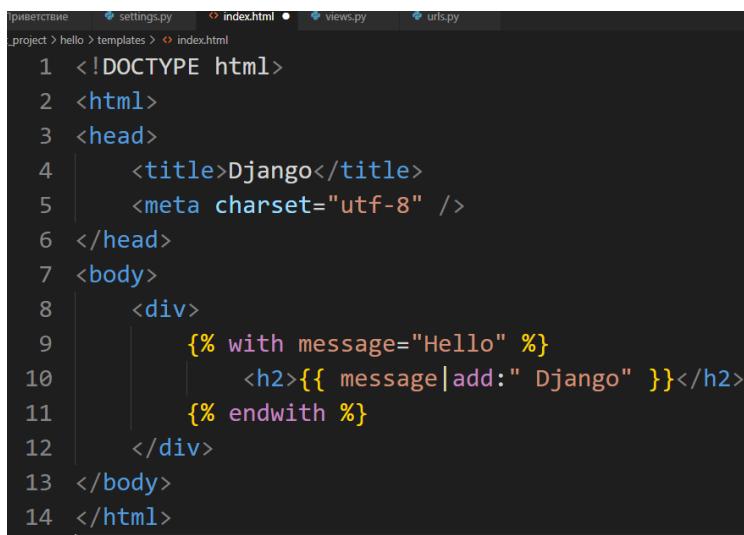
В дополнение к тегам Django также поддерживает фильтры, которые позволяют произвести некоторую простешую обработку значений внутри шаблонов. Полный список фильтров можно найти в [документации](#). Здесь же рассмотрим только некоторые наиболее используемые.

### add

Фильтр **add** добавляет к одному значению другое.

```
значение1 | add: значение2
```

Например:



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <div>
9         {% with message="Hello" %}
10            <h2>{{ message|add:" Django" }}</h2>
11        {% endwith %}
12     </div>
13 </body>
14 </html>
```

В данном случае определяется переменная `message` (это также могли бы быть данные, передаваемые из представления). Затем при выводе значения этой переменной к ней добавляется строка " Django"



### capfirst

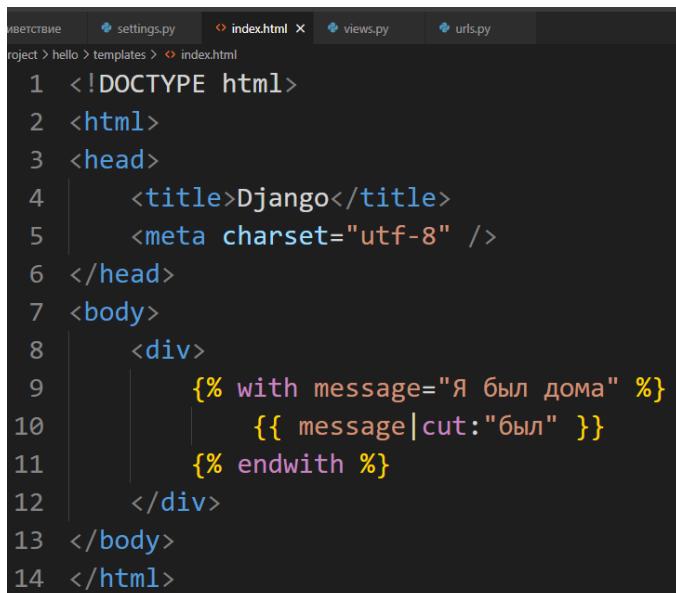
Фильтр **capfirst** делает первую букву заглавной:

```
{{ message|capfirst }}ccc
```

Например, если `message` равно "django", то после применения фильтра на веб-странице будет выведено "Django".

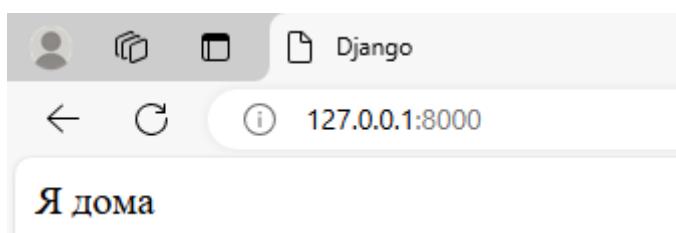
## cut

Фильтр **cut** удаляет из строки определенную подстроку:



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <div>
9         {% with message="Я был дома" %}
10            {{ message|cut:"был" }}
11        {% endwith %}
12     </div>
13 </body>
14 </html>
```

В данном случае из строки "Я был дома" удаляется подстрока "был", поэтому шаблон выведет "Я дома"



## Проверка значения и значение по умолчанию

Фильтр **default** проверяет значение, и если оно равно False, то возвращает некоторое значение по умолчанию.

```
{% with isEnabled=False %}
    {{ isEnabled|default:"unabled" }}
{% endwith %}
```

В данном случае переменная `isEnabled` равна False, поэтому шаблон выведет "unabled"

При этом сама переменная необязательно должна хранить именно True или False, это может быть любое выражение, которое можно привести к значению True/False. Например:

```
{% with name="" %}
    {{ name|default:"Undefined" }}
{% endwith %}
```

В данном случае переменная `name` представляет пустую строку, поэтому при применении фильтра получим False, а фильтр возвратит строку "Undefined".

Похожим действием обладает фильтр **default\_if\_none** - он возвращает значение по умолчанию, если переданное ему значение равно **None**:

```
{% with user=None %}
    {{ user|default_if_none:"Undefined" }}
{% endwith %}
```

### floatformat

**floatformat** форматирует числа с плавающей точкой.

Если фильтру не передается аргумент, то число округляется до одного знака после запятой:

Значение	Шаблон	Результат
34.23234	{{ value floatformat }}	34.2
34.00000	{{ value floatformat }}	34
34.26000	{{ value floatformat }}	34.3

Фильтру можно передать числовой аргумент, который указывает, до скольких знаков после запятой надо округлять:

Значение	Шаблон	Результат
34.23234	{{ value floatformat:3 }}	34.232
34.00000	{{ value floatformat:3 }}	34.000
34.26000	{{ value floatformat:3 }}	34.260

Если аргумент - отрицательное число, то число также округляется до определенного количества знака после запятой. Однако если дробная часть состоит из одних нулей, то нули при этом отсекаются:

Значение	Шаблон	Результат
34.23234	{{ value floatformat:"-3" }}	34.232
34.00000	{{ value floatformat:"-3" }}	34
34.26000	{{ value floatformat:"-3" }}	34.260

Аргумент 0 позволяет округлить число до ближайшего целого:

Значение	Шаблон	Результат
34.23234	{{ value floatformat:"0" }}	34
34.00000	{{ value floatformat:"0" }}	34
39.56000	{{ value floatformat:"0" }}	40

Если аргумент имеет суффикс **g**, то применяется группировка с использованием разделителя разрядов в соответствии с текущей локалью:

Значение	Шаблон	Результат
34232.34	<code>{{ value floatformat:"2g" }}</code>	34,232.34
34232.06	<code>{{ value floatformat:"g" }}</code>	34,232.1
34232.00	<code>{{ value floatformat:"-3g" }}</code>	34,232

При выводе применяется текущая локаль проекта. Например, если мы откроем файл **settings.py**, то можем там найти переменную **LANGUAGE\_CODE**, которая определяет локаль:

```
LANGUAGE_CODE = 'en-us'
```

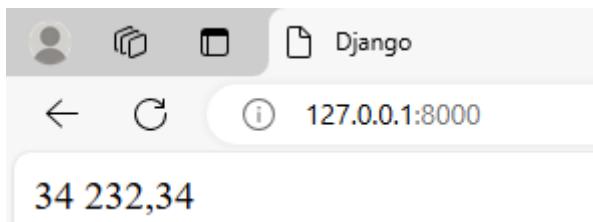
По умолчанию применяется американский вариант английского языка и соответствующие им настройки локали. Соответственно разделителем между целой и дробной частью будет точка, а между разрядами - запятая. Но, например, изменим на русскоязычную культуру:

```
LANGUAGE_CODE = 'ru-ru'
```

Теперь для кода

```
{% with value=34232.34 %}  
| {{ value|floatformat:"2g" }}  
{% endwith %}
```

Мы получим вывод



Если же необходимо отключить локализацию, аргументу добавляется суффикс **u**:

```
{{ value|floatformat:"2u" }}
```

### Форматирование дат

Фильтр **date** применяется к объекту **datetime** и в качестве аргумента получает следующие спецификаторы даты и времени:

- **d**: день месяца в виде двух цифр (одна цифра дополняется нулем слева). Диапазон значений от '01' до '31'
- **j**: день месяца без дополнения нулем. Диапазон значений от '1' до '31'
- **D**: день недели в текстовом виде в виде трех символов, например, 'Fri'
- **I**: день недели в полном виде, например, 'Friday'

- **S:** возвращает английский суффикс для порядковых числительных для дня месяца. Возможные значения: 'st', 'nd', 'rd' и 'th'
- **w:** номер дня недели. Значения в диапазоне от '**0**' (воскресенье) до '**6**' (суббота)
- **z:** номер дня года. Диапазон значений от **1** до **366**
- **W:** номер недели года (первым днем недели считается понедельник). Значения от **1** до **53**
- **m:** номер месяца в виде двух цифр. Диапазон значений от '**01**' до '**12**'
- **n:** номер месяца в виде двух цифр без предварения нулем. Диапазон значений от '**1**' до '**12**'
- **M:** текстовое значение месяца в виде трех символов. Например, '**Jan**'
- **b:** текстовое значение месяца в виде трех символов в нижнем регистре. Например, '**jan**'
- **E:** локализованное название месяца. Например, '**августа**' ("август" в родительном падеже)
- **F:** полное название месяца. Например, '**January**'
- **N:** аббревиатура месяца. Например, '**Jan.**', '**Feb.**', '**March**', '**May**'
- **t:** число дней в текущем месяце. Значения от **28** до **31**
- **y:** двухчисловой код года с дополнением нулем. Диапазон значений от '**00**' до '**99**'
- **Y:** четырехчисловой код года с дополнением нулем. Диапазон значений от '**0001**' до '**9999**'
- **L:** является ли год високосным. Значения: **True**(является) и **False** (не является)
- **o:** ISO-8601 week-numbering year,
- **g:** час в 12-часовом формате без дополнения нулем. Диапазон значений от '**1**' до '**12**'
- **G:** час в 24-часовом формате без дополнения нулем. Диапазон значений от '**0**' до '**23**'
- **h:** час в 12-часовом формате с дополнением нулем. Диапазон значений от '**01**' до '**12**'
- **H:** час в 24-часовом формате с дополнением нулем. Диапазон значений от '**00**' до '**23**'
- **i:** минуты от '**00**' до '**59**'
- **s:** секунды от '**00**' до '**59**'
- **u:** микросекунды от **000000** до **999999**
- **a:** '**a.m.**' или '**p.m.**'
- **A:** '**AM**' или '**PM**'
- **f:** время в 12-часовом формате с минутами. Например, '**1:30**'

- **P**: время в 12-часовом формате с минутами и "а.м."/ "п.м.". Например, '**12:30 p.m.**'
- **e**: часовая зона, например, '', '**GMT**', '**-500**', '**US/Eastern**', etc.
- **O**: смещение в часах относительно Гринвича. Например, '**+0200**'
- **T**: временная зона текущего компьютера. '**EST**', '**MDT**'
- **c**: дата и время в формате ISO 8601. Например, **2008-01-02T10:30:00.000123+02:00** или **2008-01-02T10:30:00.000123**
- **r**: дата и время в формате RFC 5322

Также можно передать ряд предустановленных констант:

- **DATE\_FORMAT**
- **DATETIME\_FORMAT**
- **SHORT\_DATE\_FORMAT**
- **SHORT\_DATETIME\_FORMAT**

Например, в представлении в шаблон передается некоторая дата:

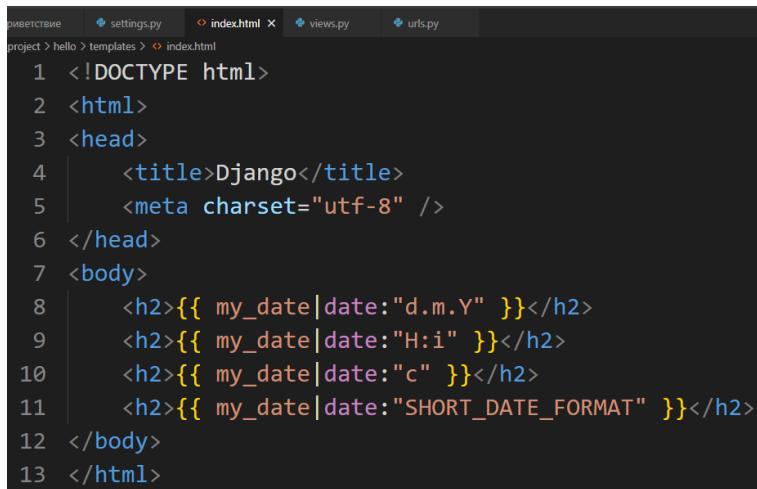
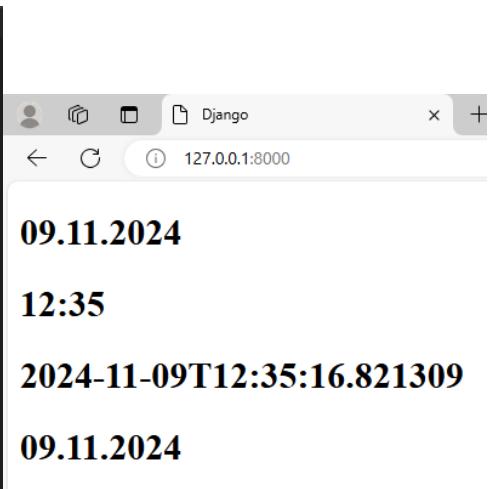


```

1 from datetime import datetime
2 from django.shortcuts import render
3
4 def index(request):
5     return render(request, "index.html", context={"my_date": datetime.now()})

```

Выведем эту дату в шаблоне:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     <h2>{{ my_date|date:"d.m.Y" }}</h2>
9     <h2>{{ my_date|date:"H:i" }}</h2>
10    <h2>{{ my_date|date:"c" }}</h2>
11    <h2>{{ my_date|date:"SHORT_DATE_FORMAT" }}</h2>
12 </body>
13 </html>

```

## Операции со списками

Ряд фильтров предназначены для работы со списками.

Фильтр **join** объединяет элементы списка, используя определенный разделитель:

```
{% users|join:", "%}
```

The screenshot shows a Django project structure with files: settings.py, index.html, views.py, and urls.py. The views.py file contains a function 'index' that returns a rendered template with a context variable 'users'. The index.html template uses a 'join' filter to concatenate the 'users' list into a single string separated by commas. The browser window shows the rendered output: 'Tom, Sam, Bob, Mike'.

```

1 from datetime import datetime
2 from django.shortcuts import render
3
4 def index(request):
5     users = ["Tom", "Sam", "Bob", "Mike"]
6     return render(request, "index.html", context={"users": users})

```

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     {{ users|join:", "}}
9 </body>
10 </html>

```

Предположим, что здесь `users` - это список `["Tom", "Sam", "Bob", "Mike"]`, то после применения фильтра получится строка `"Tom, Sam, Bob, Mike"`.

Фильтр `slice` получает часть списка. Для извлечения части списка он получает начальный и конечный индексы для извлечения элементов:

```

{{users|slice:"start"}}
{{users|slice:"end"}}
{{users|slice:"start:end"}}

```

параметр `start` указывает на индекс элемента, начиная с которого надо скопировать элементы, а через параметр `end` передается индекс элемента, до которого нужно скопировать список. Если `start` или `end` не указываются, то `start` по умолчанию равен 0, а `end` - на длину списка. Например

The screenshot shows a code editor for the index.html template. It includes a line of code using the `slice` filter to extract elements from index 1 to 3 of the 'users' list.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Django</title>
5     <meta charset="utf-8" />
6 </head>
7 <body>
8     {{ users|slice:"1:3" }}
9 </body>
10 </html>

```

Здесь извлекаются элементы с 1 по 3 индекс (не включая), то есть получится список ["Sam", "Bob"].

A screenshot of a web browser window. The address bar shows '127.0.0.1:8000'. The main content area displays the list: ['Sam', 'Bob'].

Фильтр **length** возвращает длину списка (также этот фильтр можно применять для нахождения длины строки).

A screenshot of a web browser window. The address bar shows '127.0.0.1:8000'. The main content area displays the number: 4.

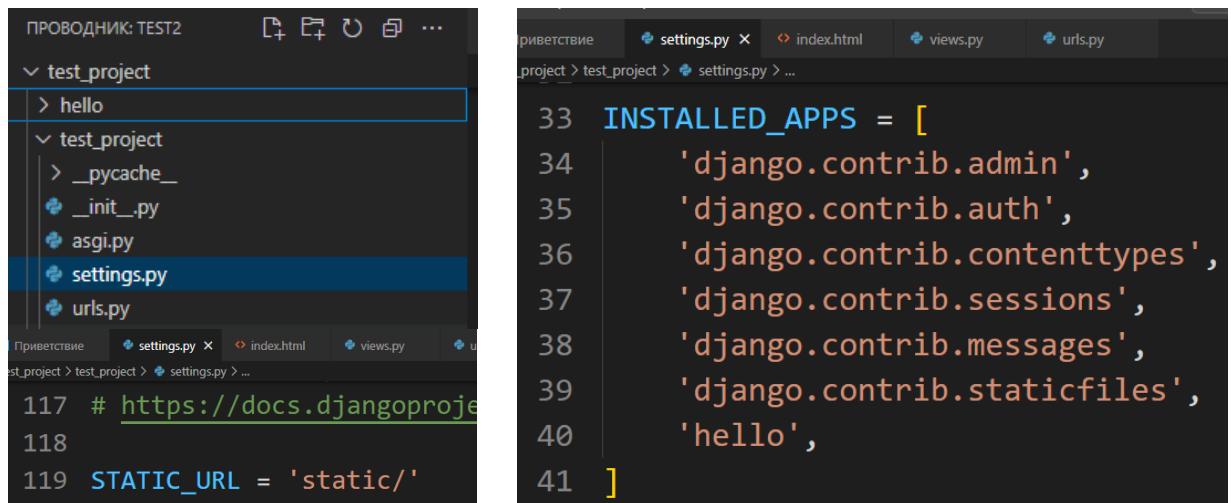
## Статические файлы

Веб-приложение, как правило, использует различные статические файлы - изображения, файлы стилей css, скриптов javascript и так далее. Рассмотрим, как мы можем использовать подобные файлы.

При создании проекта Django он уже имеет некоторую базовую настройку для работы со статическими файлами. В частности, в файле **settings.py** определена переменная **STATIC\_URL**, которая хранит путь к каталогу со статическими файлами:

```
STATIC_URL = 'static/'
```

А среди установленных приложений в переменной **INSTALLED\_APPS** указано приложение **django.contrib.staticfiles**



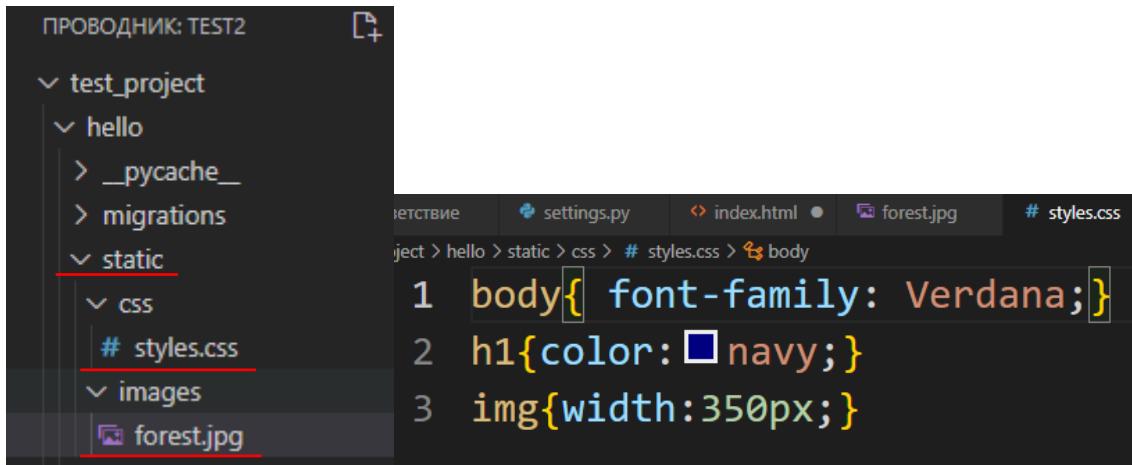
The screenshot shows a code editor with two panes. The left pane is a file browser titled 'ПРОВОДНИК: TEST2' showing a project structure with a 'test\_project' folder containing 'hello' and 'urls.py'. The right pane is a code editor with the following content:

```
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'hello',
41 ]
```

Переменная **STATIC\_URL** имеет значение "static/", а это значит, что нам достаточно создать в папке приложения каталог с именем "static" и добавить в него необходимые нам статические файлы. Но, естественно, при необходимости через данную настройку мы можем изменить расположение каталога статических файлов.

Итак, добавим в **папку приложения** новый каталог **static**. Чтобы не сваливать все статические файлы в кучу, определим для каждого типа файлов отдельные папки. В частности, создадим в папке static для изображений каталог **images**, а для стилей - каталог **css**. Подобным образом можно создавать папки и для других типов файлов.

В папку **static/images** добавим какое-нибудь изображение - в моем случае это будет файл **forest.jpg**. А в папке **static/css** определим новый файл **styles.css**, который будет иметь какие-нибудь простейшие стили, например:



Теперь используем эти файлы в шаблоне. Для этого в начале файла шаблона необходимо определить инструкцию

```
{% load static %}
```

При этом данный код должен идти после тега DOCTYPE.

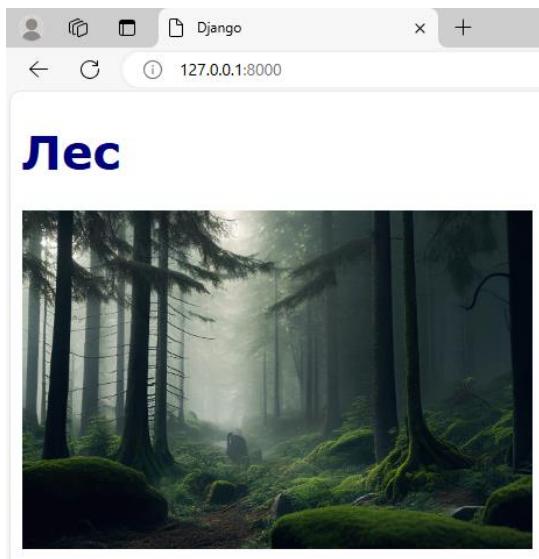
Для определения пути к статическим файлам используются выражения типа

```
{% static "путь к файлу внутри папки static" %}
```

Так, пусть в приложении в папке templates определен шаблон index.html, который имеет следующий код:

```
index.html  
---  
1 <!DOCTYPE html>  
2 {% load static from staticfiles %}  
3 <html>  
4 <head>  
5     <meta charset="utf-8" />  
6     <link rel="stylesheet" href="{% static "css/styles.css" %}" />  
7     <title>Django</title>  
8 </head>  
9 <body>  
10    <h1>Лес</h1>  
11      
12 </body>  
13 </html>
```

При запуске приложения шаблон index.html будет генерироваться в следующую веб-страницу, которая будет использовать изображение и применять стили:



## Настройка путей к файлам

Если нас не устраивает хранение файлов в каталоге по умолчанию - каталоге static, либо мы хотим указать несколько папок, то мы можем в файле **settings.py** задать все необходимые каталоги с помощью переменной **STATICFILES\_DIRS**, которая принимает список путей:

```
settings.py x index.html views.py urls.py
test_project > test_project > settings.py > ...
118
119 STATIC_URL = [ BASE_DIR / 'static/',
120 |           |   "/var/www/static/",
121 |           |   "/somefolder/"
122 ]
```