

TRẦN TUẤN DŨNG



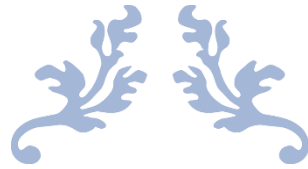
TÀI LIỆU HƯỚNG DẪN THỰC HÀNH PHÁT TRIỂN ỨNG DỤNG WEB VỚI PYTHON, DJANGO VÀ POSTGRESSQL

WEB DEVELOPMENT WITH DJANGO FRAMEWORK



Ho Chi Minh, 2021

TRẦN TUẤN DŨNG



TÀI LIỆU HƯỚNG DẪN THỰC HÀNH PHÁT TRIỂN ỨNG DỤNG WEB VỚI PYTHON, DJANGO VÀ POSTGRESSQL

WEB DEVELOPMENT WITH DJANGO FRAMEWORK



Ho Chi Minh, 2021

LỜI NÓI ĐẦU

Ngày nay, có rất nhiều ngôn ngữ lập trình và framework có thể hỗ trợ để phát triển một ứng dụng web. Trong đó, sự kết hợp giữa Python và Django đang nổi lên là một trong những ngôn ngữ lập trình và web framework phổ biến, được nhiều cá nhân và doanh nghiệp sử dụng. Ưu điểm của nó là: mạnh mẽ, nhanh chóng, tiện lợi và an toàn.

Tài liệu này giúp chúng ta có thể nhanh chóng tiếp cận ngôn ngữ lập trình Python, cũng như cách sử dụng Django framework để xây dựng một ứng dụng web hoàn chỉnh và cách triển khai dự án web trên môi trường thật.

MỤC LỤC

CHƯƠNG 1: CƠ BẢN VỀ PYTHON	1
1.1. Kiến thức cơ bản về lập trình Python cho người mới bắt đầu.....	1
1.1.1. Cài đặt Python trên Window.....	1
1.1.1.1. Cài đặt Python	1
1.1.1.2. Cài đặt Visual Studio Code (VS Code).....	5
1.1.2. Python Hello World.....	8
1.1.3. Hàm print() trong Python.....	10
1.1.3.1. Định nghĩa	10
1.1.4. Biến số trong Python	11
1.1.4.1. Định nghĩa	11
1.1.4.2. Cách sử dụng và ví dụ	11
1.1.4.3. Biến toàn cục (Global) và biến cục bộ (Local).....	13
1.2. Cấu trúc dữ liệu Python.....	16
1.2.1. Loại dữ liệu	16
1.2.2. Number	17
1.2.3. Python Strings.....	18
1.2.3.1. Sửa đổi chuỗi.....	18
1.2.3.2. Định dạng chuỗi	19
1.2.4. List	20
1.2.4.1. Truy xuất phần tử của List	21
1.2.4.2. Cập nhật List	22
1.2.4.3. Vòng lặp với list	24
1.2.4.4. Sort list	24
1.2.4.5. Bài tập.....	25
1.2.5. Tuple	27
1.2.5.1. Truy xuất phần tử của Tuple	28
1.2.5.2. Cập nhật Tuple	30

1.2.5.3. Giải nén Tuple	32
1.2.5.4. Vòng lặp với Tuple.....	33
1.2.5.5. Bài tập.....	34
1.2.6. Python Dictionary(Dict)	35
1.2.6.1. Truy xuất các phần tử của dictionary	36
1.2.6.2. Cập nhật dict.....	37
1.2.6.3. Vòng lặp với dict	39
1.2.6.4. Bài tập.....	40
1.3. Câu lệnh rẽ nhánh.....	40
1.4. Vòng lặp	41
1.4.1. Vòng lặp For	41
1.4.2. Vòng lặp while.....	41
1.5. Hàm trong Python	42
1.6. Kế thừa trong Python	47
CHƯƠNG 2: DJANGO.....	49
2.1. Django	49
2.1.1. Thiết lập môi trường ảo	50
2.1.1.1. Cài đặt pip	50
2.1.2. Cài đặt cơ sở dữ liệu.....	51
2.1.2.1. PostgreSQL.....	51
2.1.2.2. Lợi ích của việc sử dụng PostgreSQL với Django.....	52
2.1.2.3. Cài đặt PostgreSQL	52
2.1.3. Tiến hành cài đặt Django	53
2.1.4. Khởi tạo dự án web đầu tiên.....	53
2.1.4.1. Tạo ứng dụng đầu tiên.....	55
2.1.4.2. Cài đặt cấu hình Databases.....	56
2.2. Models và Databases.....	57
2.2.1. Models	57
2.2.1.1. Fields (Các trường).....	59
2.2.1.2. Các mối quan hệ (Relationships)	61

2.2.2. Tạo câu truy vấn	64
2.2.3. Django admin site	75
2.2.3.1. Model admin.....	75
2.3. Xử lý HTTP requests	80
2.3.1. URL dispatcher	80
2.3.1.1. Sử dụng regular expressions.....	82
2.3.1.2. Chỉ định các giá trị đối số cho view	84
2.3.1.3. Thêm các URLconfs khác	84
2.3.1.4. Captured parameters từ URL pattern	86
2.3.2. Xây dựng view	86
2.3.2.1. Trả về lỗi	87
2.3.2.2. Http404 exception	88
2.3.3. View decorators	89
2.3.4. File Uploads	90
2.3.4.1. File uploads cơ bản.....	90
2.3.4.1.1. Xử lý upload với model.....	91
2.3.4.1.2. Upload nhiều file	92
2.3.4.2. Xử lý upload	93
2.3.4.3. Các hàm rút gọn	95
2.3.4.3.1. Hàm render()	95
2.3.4.3.2. Hàm redirect()	95
2.3.5. Generic views	96
2.3.5.1. Base views	96
2.3.5.1.1. View.....	96
2.3.5.1.2. TemplateView	98
2.3.5.1.3. RedirectView	99
2.3.5.2. Generic display views	101
2.3.5.2.1. DetailView	101
2.3.5.2.2. ListView	103
2.3.5.3. Generic editing views.....	105

2.3.5.3.1. FormView	105
2.3.5.3.2. CreateView	106
2.3.5.3.3. UpdateView	107
2.3.5.3.4. DeleteView	108
2.4. Forms.....	109
2.4.1. HTML form	109
2.4.2 Django Form	110
2.4.3. Formsets	115
2.4.4 Tạo form từ các model.....	130
2.4.4.1. Model form.....	130
2.4.4.1.1. Field types	130
2.4.4.1.2. Ví dụ cụ thể	133
2.4.4.1.3. Xác thực trên ModelForm	134
2.4.4.1.4. Phương thức save().....	134
2.4.4.1.5. Ghi đè các trường mặc định	138
2.4.4.1.6. Kế thừa Form.....	142
2.4.4.2. Model formsets.....	143
2.4.4.2.1. Thay đổi queryset	144
2.4.4.2.2. Lưu các đối tượng trong formset	144
2.4.4.2.3. Giới hạn số lượng đối tượng có thể hiển thị.....	145
2.4.4.2.4. Sử dụng Model formset trong một view.....	146
2.4.4.2.5. Tùy chỉnh queryset	146
2.4.4.2.6. Sử dụng Formset trong một template	147
2.4.4.3. Inline formsets.....	149
2.4.5. Form Assets (Media class).....	151
2.4.5.1. Nội dung dưới dạng định nghĩa static	152
2.4.5.2. CSS	153
2.4.5.3. Media như một thuộc tính động:	153
2.4.5.4. Đường dẫn của các file nội dung:.....	154
2.4.5.5. Đối tượng Media	156

2.5. Template.....	160
2.6. Class-based views.....	160
2.6.1. Sử dụng Class-based views.....	161
2.6.2. Xử lý forms với class-based views	162
2.6.3. Decorate pattern trong class.....	164
2.6.4. Xây dựng trong class-based generic views.....	165
2.6.4.1. Generic views của các đối tượng.....	165
2.7. Migrations	167
2.7.1. Commands	167
2.7.2. Workflow	168
2.7.3. Transactions	169
2.7.4. Dependencies	170
2.7.5. Migrations file.....	170
2.7.5. Initial migrations.....	171
2.7.6. Thêm migration vào ứng dụng.....	171
2.7.7. Đảo ngược quá trình migration.....	172
2.8. Quản lí File.....	172
2.8.1. Sử dụng tệp trong các model	172
2.8.2. Đối tượng tệp	173
2.8.3. File storage.....	174
2.9. Signals (Tín hiệu)	175
2.9.1. Lắng nghe signals	175
2.9.1.1. Hàm receiver	175
2.9.1.2. Kết nối các hàm receiver	176
2.9.1.3. Kết nối đến tín hiệu được gửi bởi những sender cụ thể	176
2.10. User Authentication	177
2.10.1. Sử dụng hệ thống xác thực Django.....	177
2.10.1.1. Đối tượng User	177
2.10.1.2. Tạo user	177
2.10.1.3. Tạo superusers	178

2.10.1.4. Thay đổi password.....	178
2.10.1.5. Chứng thực user	179
2.10.1.6. Permissions và Authorization.....	179
2.10.1.6.1. Đăng nhập.....	180
2.10.1.6.2. Đăng xuất.....	181
2.10.1.6.3. Sử dụng chứng thực trong template	181
2.10.2. Quản lý password trong Django.....	183
2.10.2.1. Lưu trữ password.....	183
2.10.2.2. Bật xác thực mật khẩu	184
2.11. Serializing objects	185
2.11.1. Serializing data	185
2.11.1.1. Serialize tập hợp con của các trường	186
2.11.1.2. Serialize model được kế thừa	186
2.11.2. Deserializing data	187
2.11.3. Serialization formats	188
2.12. Ký mã hóa	188
2.12.1. Bảo vệ SECRET_KEY	188
2.12.2. Sử dụng các API cấp thấp.....	188
2.12.2.1. Sử dụng đối số salt	190
2.12.2.2. Xác minh thời gian của chữ ký.....	191
CHƯƠNG 3: TRIỂN KHAI ỨNG DỤNG	193
3. Tổng quan.....	193
3.1. Môi trường production là gì?	193
3.2. Chọn nhà cung cấp dịch vụ lưu trữ	194
3.3. Heroku.....	195
3.3.1. Cách thức hoạt động của Heroku.....	195
3.3.2. Setup các tập tin cần thiết cho Heroku	196
3.3.3. Tạo tài khoản Heroku	200
3.3.4. Cài đặt Heroku client	201
3.3.5. Tạo và deploy website	201

TÀI LIỆU THAM KHẢO	203
--------------------------	-----

DANH MỤC HÌNH VẼ

Hình 1. 1 Cài đặt Python	2
Hình 1. 2 Ảnh sau khi cài đặt thành công.....	3
Hình 1. 3 Kiểm tra kết quả sau khi cài đặt	3
Hình 1. 4 Search sysdm.cpl	4
Hình 1. 5 Thêm path vào system variable	4
Hình 1. 6 Download VS code.....	5
Hình 1. 7 Accept agreement	6
Hình 1. 8 Chọn next.....	6
Hình 1. 9 Chọn Install.....	7
Hình 1. 10 Cửa sổ báo đã cài đặt hoàn thành	7
Hình 1. 11 Giao diện VS code.....	8

DANH MỤC BẢNG

Bảng 2. 1 Các thuộc tính trên field.....	115
Bảng 2. 2 Danh sách chuyển đổi giữa modelfield và formfield.....	130

DANH MỤC TỪ VIẾT TẮT

Thuật ngữ thường dùng	Từ đầy đủ	Giải thích
API	Application Programming Interface	Mô tả cho một kết nối giữa những máy tính hoặc giữa các chương trình với nhau
CSDL	Cơ sở dữ liệu	Một tập hợp các dữ liệu có tổ chức, thường được dùng để lưu trữ và truy vấn
JS	Javascript	Ngôn ngữ lập trình Javascript
IDE	Integrated Development Environment	Môi trường tích hợp dùng để viết code để phát triển ứng dụng

CHƯƠNG 1: CƠ BẢN VỀ PYTHON

1.1. Kiến thức cơ bản về lập trình Python cho người mới bắt đầu

Python đã và đang là một trong những ngôn ngữ lập trình phổ biến nhất trên thế giới với các kho thư viện hỗ trợ khổng lồ. Python là một ngôn ngữ hướng đối tượng, rất dễ học và đơn giản, nhưng cũng không kém phần mạnh mẽ về mặt hiệu năng. Với python, chúng ta không chỉ có thể viết ứng dụng, phát triển web, mà còn có thể sử dụng nó vào nhiều lĩnh vực khác như: Machine Learning, Deep Learning, AI,...

1.1.1. Cài đặt Python trên Window

1.1.1.1. Cài đặt Python

Bước 1: Chọn phiên bản Python để cài đặt:

Truy cập vào trang <https://www.python.org/downloads/> chọn phiên bản mong muốn và ấn download. Hiện tại Python có 2 phiên bản là 2.x và 3.x, tuy nhiên phiên bản 2.x đã khá cũ và sắp tới sẽ không được hỗ trợ nữa, do đó chúng ta ưu tiên cài đặt python 3.x

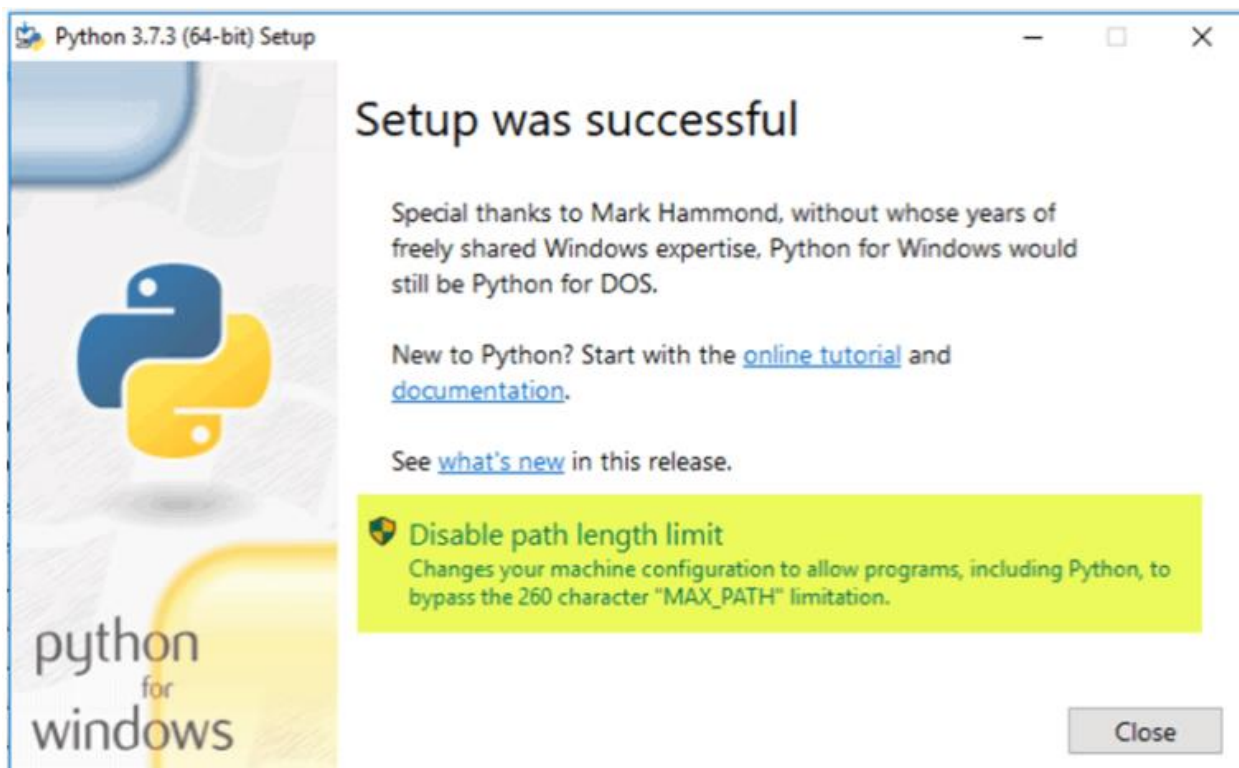
Bước 2: Chạy trình cài đặt của Python:

- Sau khi hoàn thành việc tải về file installer ta tiến hành chạy file này
- Chọn vào check box **Install launcher for all users** và **Add Python x.y. to PATH**
- Chọn **Install now**



Hình 1. 1 Cài đặt Python

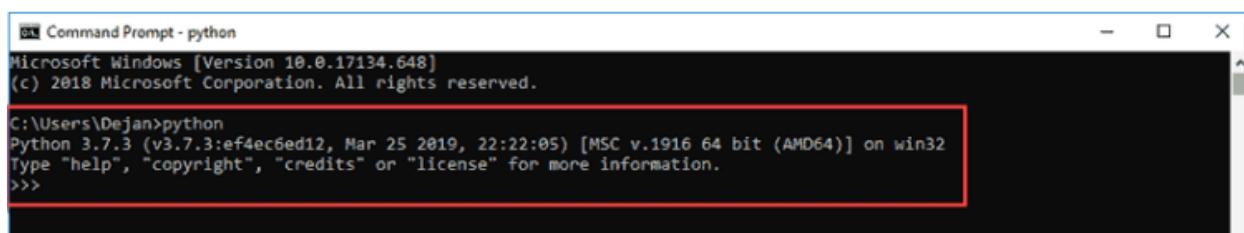
- Ở hộp thoại tiếp theo ta chọn Disable path length limit



Hình 1. 2 Ảnh sau khi cài đặt thành công

Bước 3: Xác minh Python đã được cài đặt trên Windows

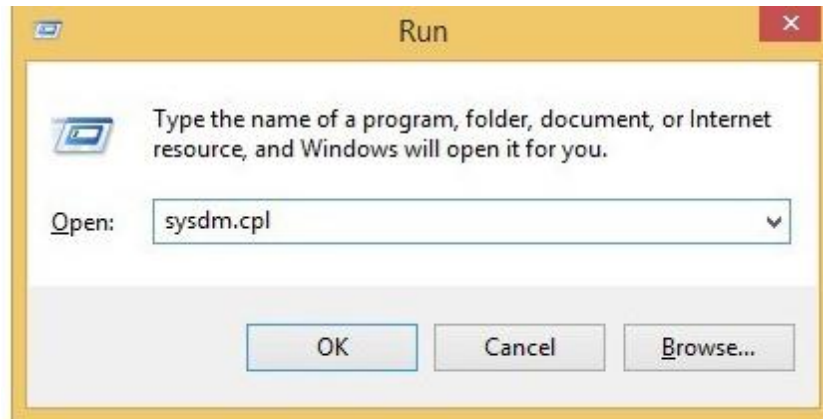
- Điều hướng đến thư mục mà Python đã được cài đặt trên hệ thống. Trong trường hợp của ví dụ, đó là "C:\Users\Username\AppData\Local\Programs\Python\Python37".
- Chạy file **python.exe**
- Kết quả sẽ như sau (phiên bản được cài đặt là 3.7.3):



Hình 1. 3 Kiểm tra kết quả sau khi cài đặt

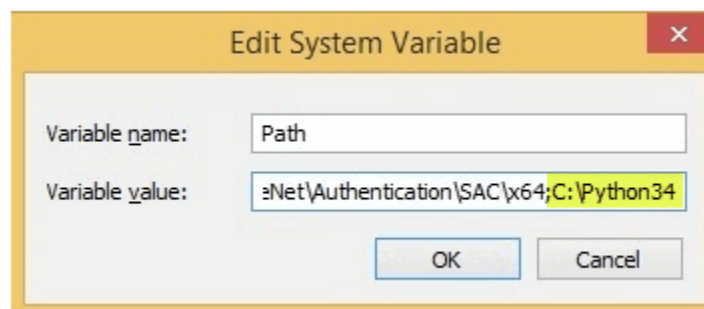
Bước 4: Thêm đường dẫn Python vào các biến môi trường

Mở **Start** menu sau đó chọn **Run**:



Hình 1. 4 Search sysdm.cpl

- Nhập *sysdm.cpl* và nhấp vào OK. Thao tác này sẽ mở ra cửa sổ System Properties thống.
- Điều hướng đến tab Advanced và chọn Environment Variables.
- Trong System Variables, hãy tìm và chọn Path variable.
- Nhấp vào Edit.
- Chọn trường Variable value. Thêm đường dẫn đến tệp python.exe trước bằng dấu chấm phẩy (;). Ví dụ: trong hình ảnh bên dưới, "; C:\Python34."



Hình 1. 5 Thêm path vào system variable

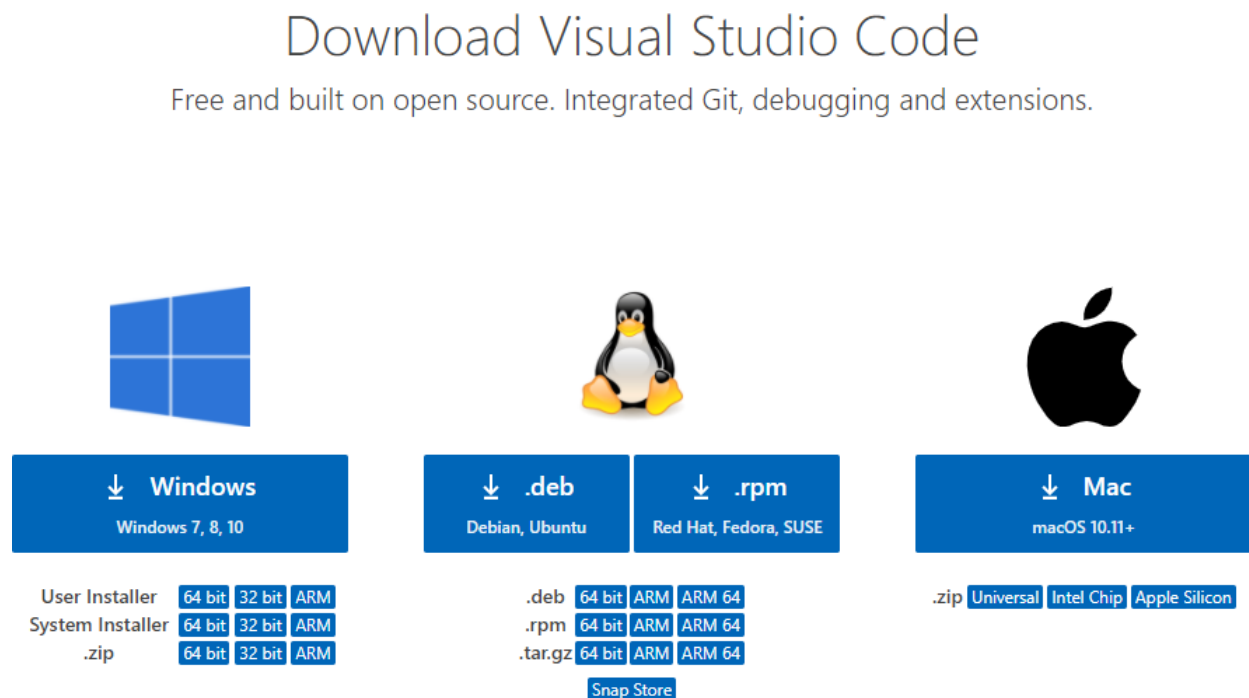
- Ấn ok

1.1.1.2. Cài đặt Visual Studio Code (VS Code)

VS Code là một IDE do Microsoft phát triển, hoàn toàn miễn phí và dễ sử dụng. Đây là một công cụ không thể thiếu đối với các lập trình viên. Ưu điểm của nó là: nhẹ, nhiều tính năng.

Download Visual Studio Code:

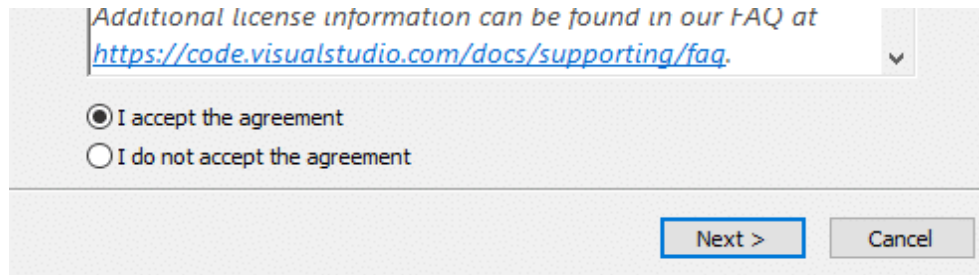
Chúng ta có thể tải xuống Visual Studio Code từ địa chỉ: <https://code.visualstudio.com/download>, chương trình với sự hỗ trợ nhiều nền tảng hệ điều hành:



Hình 1. 6 Download VS code

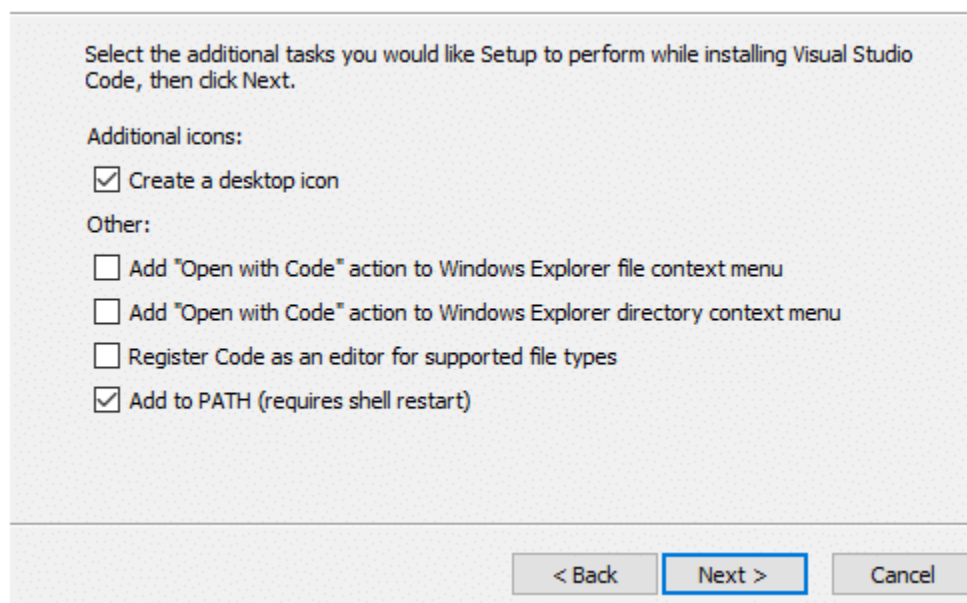
Install Visual Studio Code trên Windows:

- Sau khi tải về vscode installer, ta tiến hành chạy installer để cài đặt



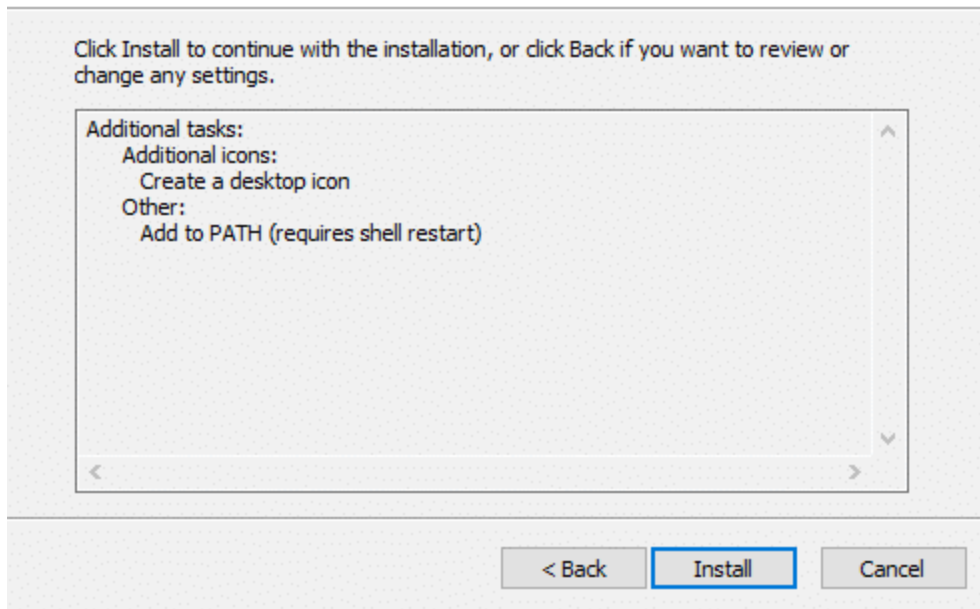
Hình 1. 7 Accept agreement

- Chọn **accept the agreement**, ấn **next**

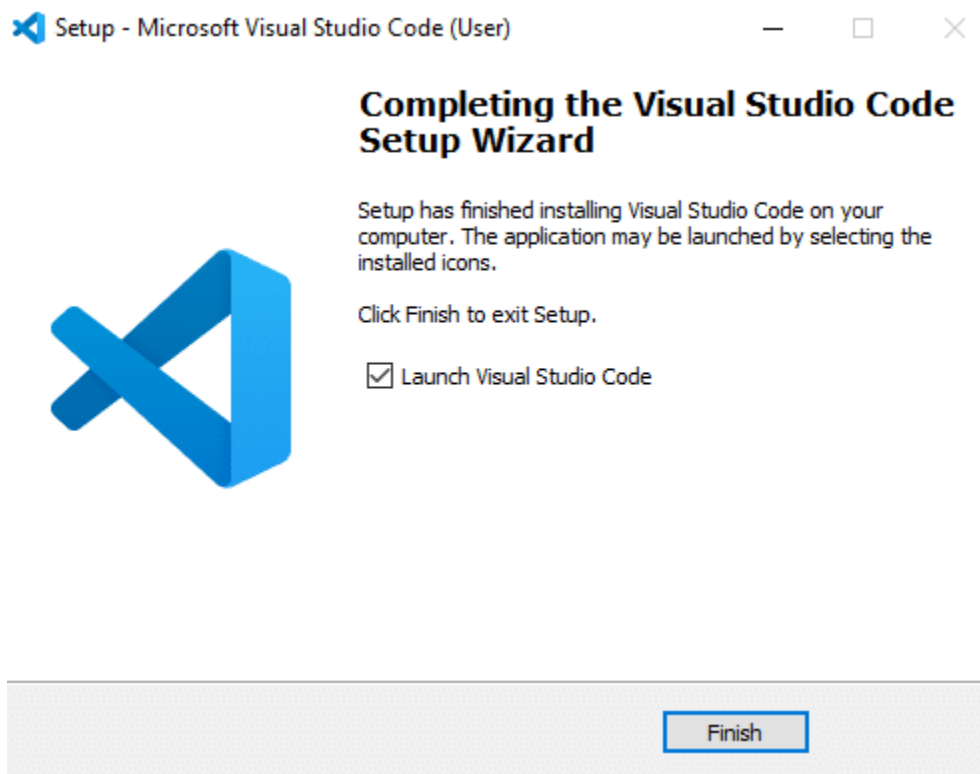


Hình 1. 8 Chọn next

- Tiếp đến click Install

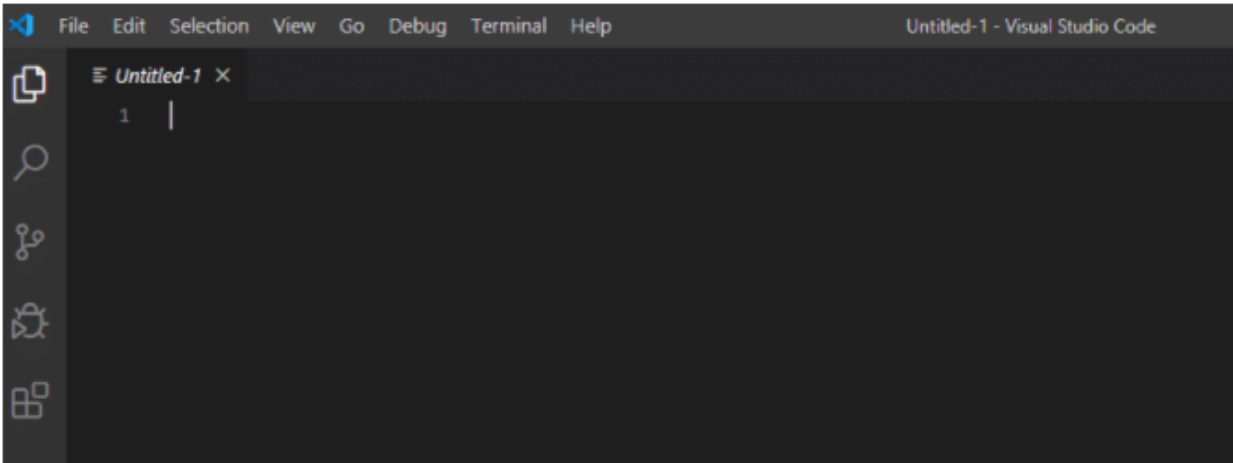


Hình 1. 9 Chọn Install



Hình 1. 10 Cửa sổ báo đã cài đặt hoàn thành

- Kết quả sau khi hoàn thành việc cài đặt VS Code



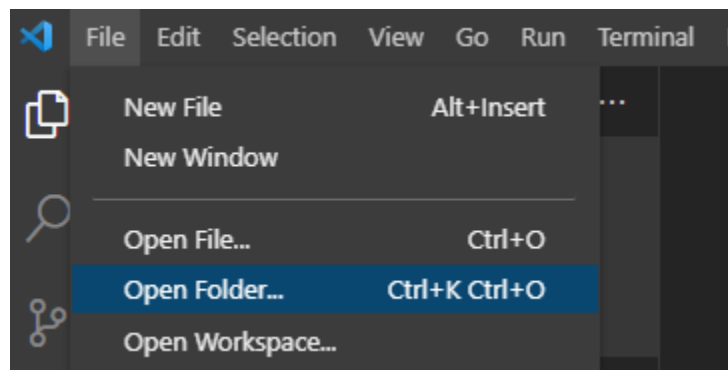
Hình 1.11 Giao diện VS code

Chọn **Create desktop a icon**, click **next** (để tạo shortcut)

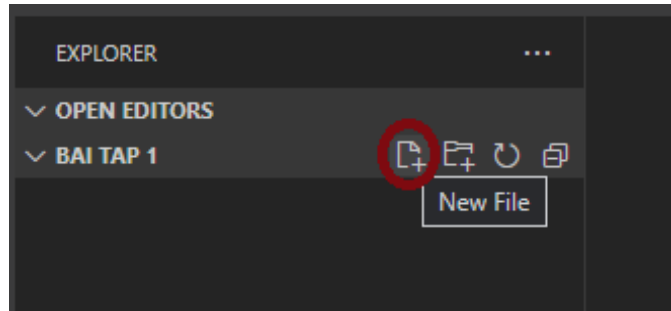
1.1.2. Python Hello World

Tạo chương trình đầu tiên:

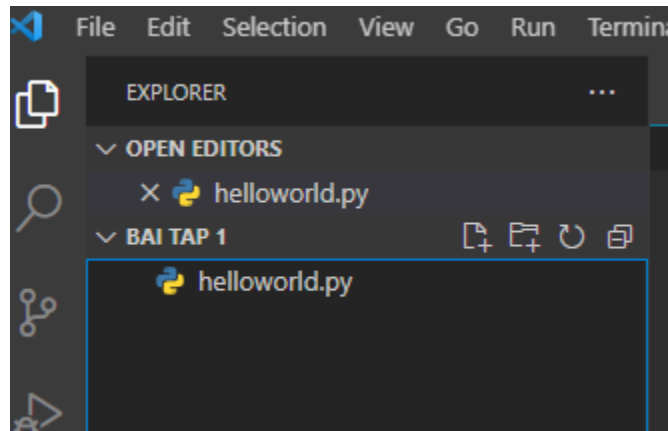
Bước 1: Mở **VS Code**, chọn **File** → **Open Folder** sau đó chọn thư mục sẽ được sử dụng cho chương trình đầu tiên.



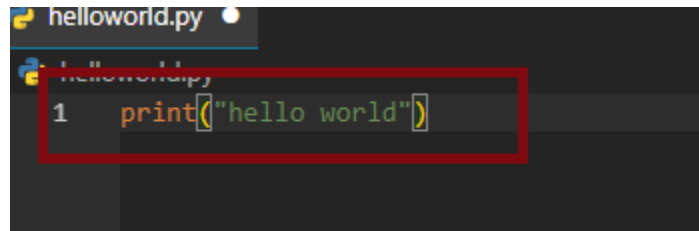
Bước 2: Bên trái màn hình quản lí, chọn thư mục với tên trùng với tên thư mục đã mở trước đó, chọn “**New File**”



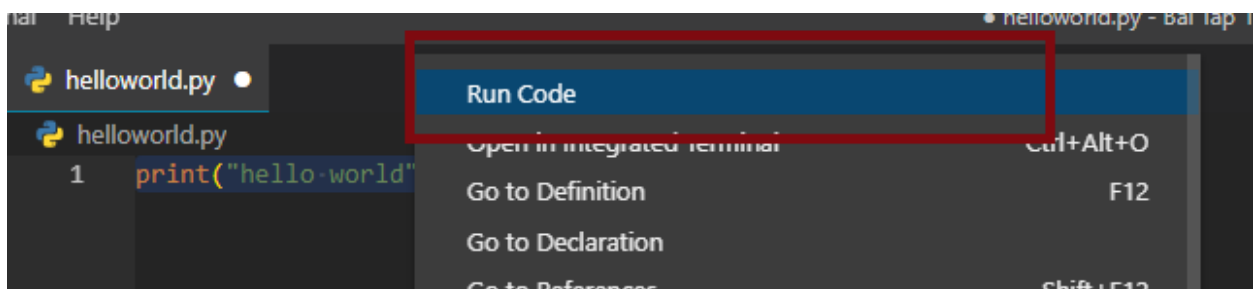
Bước 3: Tạo file với tên gọi **helloworld.py**



Bước 4: Bên khung màn hình soạn thảo nhập đoạn mã sau: **print("hello world")**



Bước 5: Bôi đen đoạn mã → click chuột phải sau đó chọn **“Run Code”**



Chúng ta có thể nhìn thấy kết quả của chương trình ở màn hình bên dưới:

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
PS D:\Work\Giao Trinh\Huong Dan\Ly Thuyet\Bai Tap 1> python -u
hello world
PS D:\Work\Giao Trinh\Huong Dan\Ly Thuyet\Bai Tap 1> 
```

1.1.3. Hàm print() trong Python

1.1.3.1. Định nghĩa

Hàm print () trong Python được sử dụng để in một thông báo được chỉ định trên màn hình. Lệnh in trong Python in các chuỗi hoặc đối tượng được chuyển đổi thành chuỗi trong khi in trên màn hình.

1.1.3.2. Ví dụ

Cú pháp như sau:

```
print(object(s))
```

Ví dụ 1: Để in ra màn hình dòng “**hello world**” ta thực hiện cú pháp sau:

```
print("hello world")
```

Ví dụ 2: In ra màn hình các dòng trống

```
print(8*"\n")
```

hoặc

```
print("\n\n\n\n\n\n\n\n\n")
```

Code ví dụ:

```
print("hello world")
```

```
print(8*"\n")
```

```
print("hello world")
```

1.1.4. Biến số trong Python

1.1.4.1. Định nghĩa

Biến trong Python là một vị trí bộ nhớ dành riêng để lưu trữ các giá trị. Nói cách khác, một biến trong chương trình python cung cấp dữ liệu cho máy tính để xử lý.

Mọi giá trị trong Python đều có một kiểu dữ liệu. Các kiểu dữ liệu khác nhau trong Python ví dụ: Object, Numbers, List, Tuple, Strings, Dictionary, v.v. Các biến trong Python có thể được khai báo theo luật sau:

- Biến trong python có thể là số hoặc chữ, nhưng không được bắt đầu bằng số.
- Có thể bắt đầu tên biến bằng ký tự `_` hoặc chữ viết thường, không nên dùng chữ viết hoa để bắt đầu tên biến.
- Không thể dùng ký tự đặc biệt trong tên biến như: `!`, `@`, `#`, `$`, `%`, `*`...
- Không dùng các python keywords để đặt tên biến, ví dụ: `class`, `def`, `return`, `try`, `pass`,...

1.1.4.2. Cách sử dụng và ví dụ

Khai báo và sử dụng một Biến:

```
a = 100  
print(a)
```

Khai báo lại một biến:

- Chúng ta có thể khai báo lại các biến Python ngay cả sau khi biến đã khai báo một lần.
- Ở đây chúng ta có Python khai báo biến được khởi tạo thành **f = 0**.
- Sau đó, ta gán lại biến f thành giá trị **"django"**


```
# khởi tạo và khai báo biến.
```

```
a = 100
```

```
print(a)
```

```
# khởi tạo lại biến
```

```
a = "hello world"
```

```
print(a)
```

Kết nối chuỗi và biến trong Python

- Chúng ta xem xét việc có thể nối các kiểu dữ liệu khác nhau như chuỗi và số với nhau hay không. Ví dụ: ghép **"django"** với số **"99"**.

```
print("django" + 99)
```

- Không giống như Java, nối số với chuỗi mà không khai báo số dưới dạng chuỗi, trong khi khai báo các biến trong Python yêu cầu khai báo số dưới dạng chuỗi nếu không lỗi `TypeError` sẽ hiển thị lỗi như sau:

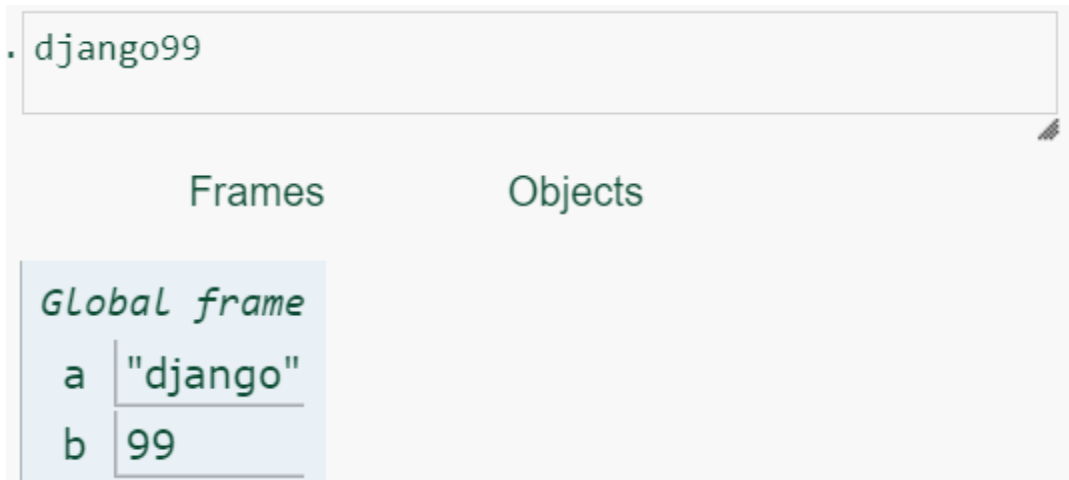
```
print("django" + 99)
TypeError: can only concatenate str (not "int") to str
```

- Khi số nguyên được ép kiểu dạng chuỗi, ta có thể nối **"django"** với **"99"**

```
a = "django"
```

```
b = 99
```

```
print(a + str(b))
```



1.1.4.3. Biến toàn cục (Global) và biến cục bộ (Local)

Có hai loại biến trong Python

- *Biến toàn cục:*
 - Các biến được tạo bên ngoài một hàm được gọi là các biến toàn cục.
 - Các biến toàn cục có thể được sử dụng ở bất cứ đâu, cả bên trong hàm và bên ngoài.
- *Biến cục bộ:*
 - Biến cục bộ được tạo bên trong hàm.
 - Nếu tạo một biến có cùng tên bên trong một hàm, thì biến này sẽ là cục bộ và chỉ có thể được sử dụng bên trong hàm. Biến toàn cục có cùng tên sẽ vẫn như cũ, toàn cục và với giá trị ban đầu.

Từ khóa **global**

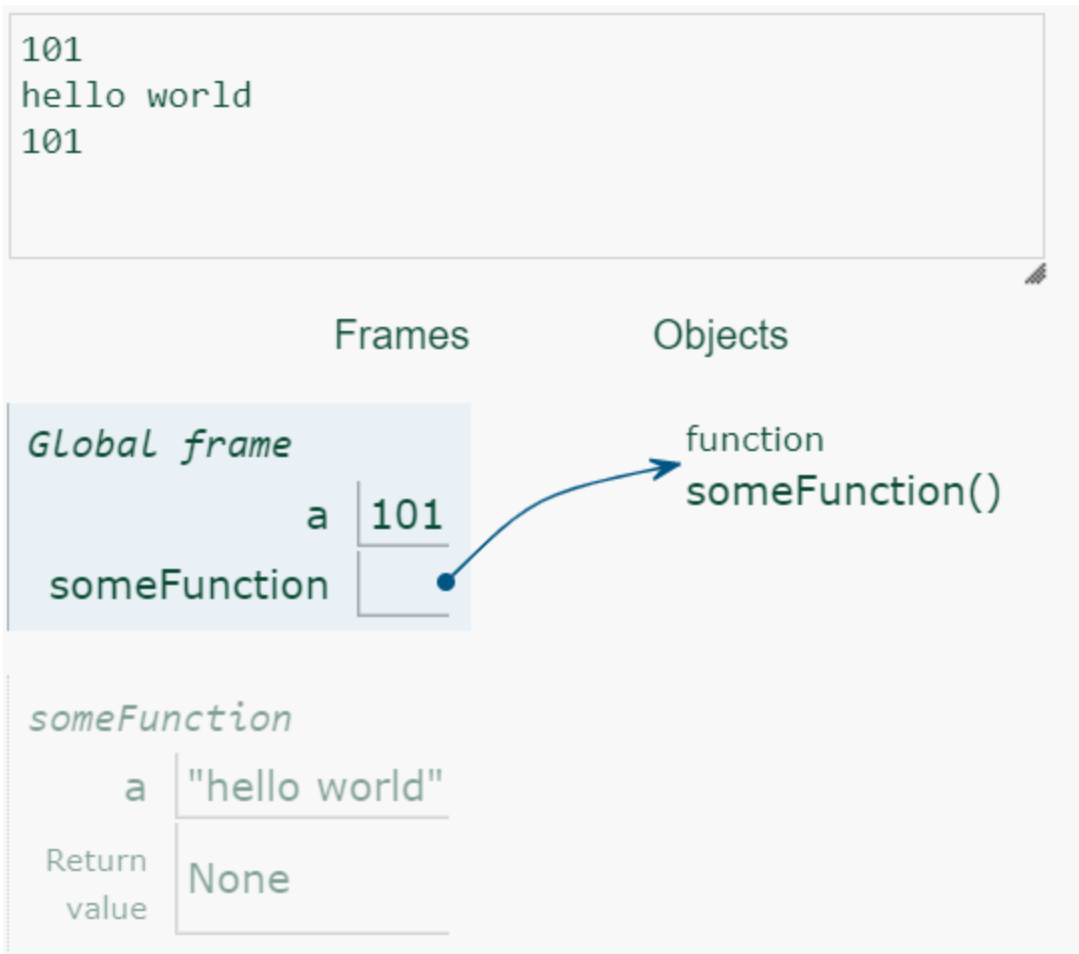
- Khi chúng ta tạo một biến bên trong một hàm, biến đó là cục bộ và chỉ có thể được sử dụng bên trong hàm đó.
- Để tạo một biến toàn cục bên trong một hàm, chúng ta có thể sử dụng từ khóa **global**.

Sự khác biệt giữa biến cục bộ và biến toàn cục có thể được thấy rõ trong chương trình bên dưới:

```
a = 101
print(a)

def someFunction():
    a = "hello world"
    print(a)
someFunction()
print(a)
```

- Tại dòng số 4, ta xác định biến “f” được khai báo ở phạm vi **global** và được gán giá trị là 101.
- Biến “f” sau đó được khai báo lại bên trong hàm someFunction() và có phạm vi khai báo là local. Tại dòng số 9, kết quả output in ra màn hình là có giá trị là giá trị của biến local và được khai báo với giá trị là **“hello world”**. Tại dòng số 11, kết quả in ra màn hình của biến “f” sẽ mang giá trị của biến “f” global (toàn cục) được khai báo trước đó.

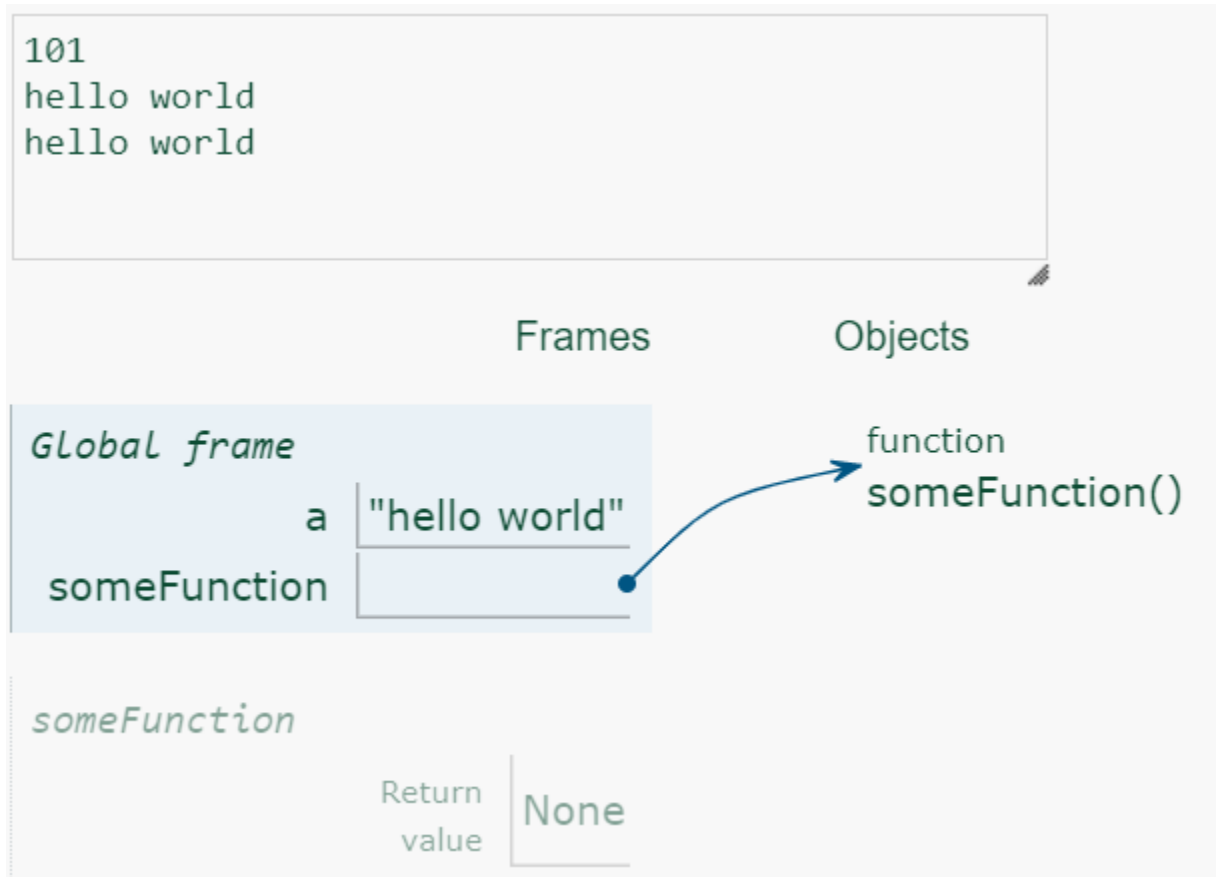


*Biến sau khi được khai báo ở phạm vi **global** có thể được thay đổi bên trong hàm.*

```
a = 101
print(a)

def someFunction():
    global a
    a = "hello world"
    print(a)
someFunction()
print(a)
```

- Biến “f” được khai báo ở phạm vi toàn cục và được gán giá trị ban đầu là 101.
- Tại dòng số 8 biến “f” được khai báo với từ khóa **global** nghĩa là biến “f” được sử dụng trong hàm là biến “f” toàn cục đã được khai báo trước đó. Tại dòng số 9, kết quả in ra màn hình sẽ là kết quả của biến toàn cục “f” hay 101.
- Sau khi thay đổi giá trị của biến “f” toán cục thành “**hello world**” bên trong hàm, thì khi lệnh gọi hàm kết thúc, giá trị của biến toàn cục “f” đã thay đổi. Tại dòng 12, khi chúng ta in lại giá trị của “f” thì kết quả hiển thị sẽ là “**hello world.**”.



1.2. Cấu trúc dữ liệu Python

1.2.1. Loại dữ liệu

Theo mặc định Python phân loại dữ liệu gồm các nhóm sau:

Nhóm	Loại dữ liệu
------	--------------

Binary	bytes, bytearray, memoryview
Text	str
Numeric	int, float, complex
Sequence	list, tuple, range
Mapping	dict
Set	set, frozenset
Boolean	bool

Để lấy loại dữ liệu từ dữ liệu có sẵn ta sử dụng hàm **type()**:

```
x = 5
print(type(x))
```

1.2.2. Number

Có ba loại dữ liệu dạng số trong Python:

- int
- float
- complex

Ví dụ:

```
x = 1      # int
y = 2.8    # float
z = 1j     # complex
```

Kiểu **Int** (số nguyên) có thể dương hoặc âm, không có số thập phân và có độ dài không giới hạn:

```
x = 1
y = 35656222554887711
z = -3255522
```

Kiểu float (hay “Floating point number”) là một số, dương hoặc âm, chứa một hoặc nhiều số thập phân.

```
x = 1.10
y = 1.0
z = -35.59
```

1.2.3. Python Strings

Định nghĩa: Các chuỗi trong python được bao quanh bởi dấu ngoặc kép đơn hoặc dấu ngoặc kép.

Chúng ta có thể hiển thị string bằng hàm **print()**:

```
print("Hello")
print('Hello')
```

String là một chuỗi Arrays, nghĩa là chúng ta có thể truy xuất và lặp qua các phần tử của string:

```
a = "Hello, World!"
print(a[1])
```

```
for x in "String":
    print(x)
```

*Chúng ta có thể sử dụng hàm **len()** để in lấy độ dài chuỗi của string:*

```
a = "Hello, World!"
print(len(a))
```

1.2.3.1. Sửa đổi chuỗi

Chúng ta có thể sử dụng hàm **upper()** hoặc **lower()** để sửa đổi chuỗi thành chữ hoa hoặc chữ thường:

```
a = "Hello, World!"
print(a.upper())
print(a.lower())
```

Khoảng trắng là khoảng không gian giữa các ký tự và ta có thể loại bỏ khoảng trắng ở đầu và ở cuối chuỗi bằng cách sử dụng hàm strip():

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

Replace String:

*Phương thức **replace()** có thể thay thế của một string bằng chuỗi string khác.*

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Split String:

*Phương thức **split()** chia chuỗi thành các chuỗi con nếu nó tìm thấy đối tượng phân tách:*

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

1.2.3.2. Định dạng chuỗi

Python không hỗ trợ kết hợp chuỗi như sau:

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

Thay vào đó chúng ta có thể sử dụng hàm format() với đối số sẽ được format và thay thế cho vị trí chứa {}:


```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

1.2.4. List

Định nghĩa:

- List được sử dụng để lưu trữ nhiều mục trong một biến duy nhất.
- List là một trong 4 kiểu dữ liệu tích hợp sẵn trong Python được sử dụng để lưu trữ các bộ sưu tập dữ liệu, 3 kiểu còn lại là Tuple, Set và Dictionary, tất cả đều có tính chất và cách sử dụng khác nhau.
- Danh sách được tạo bằng cách sử dụng dấu ngoặc vuông:

```
my_list = ["Hoc sinh", 1, False]
```

Các đặc điểm của List:

- Các mục trong **List** được sắp xếp theo thứ tự, có thể thay đổi và cho phép các giá trị trùng lặp.
- Các mục trong **List** được lập chỉ mục, mục đầu tiên có chỉ mục [0], mục thứ hai có chỉ mục [1], v.v.
- Các mục trong **List** có thứ tự xác định và thứ tự đó sẽ không thay đổi.
- Nếu thêm các mục mới vào danh sách, các mục mới sẽ được đặt ở cuối **List**.
- **List** có thể thay đổi, nghĩa là chúng ta có thể thay đổi, thêm và xóa các mục trong danh sách sau khi nó đã được tạo.
- Vì danh sách được lập chỉ mục nên danh sách có thể có các mục có cùng giá trị.
- Để xác định List có bao nhiêu mục, ta sử dụng hàm **len()**:

```
my_list = ["Hoc sinh", 1, False]
print(len(my_list))
```

- Các mục trong danh sách có thể thuộc bất kỳ kiểu dữ liệu nào:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
my_list_1 = [1, 2, 3]
my_list_2 = [True, False]
```

- Một danh sách có thể chứa các kiểu dữ liệu khác nhau:

```
my_list = ["Hoc sinh", 1, True]
```

1.2.4.1. Truy xuất phần tử của List

Các phần tử trong List được lập chỉ mục và chúng ta có thể truy cập chúng bằng cách tham chiếu số chỉ mục:

```
my_list = ["Hoc sinh", 1, True]
print(my_list[0])
```



Truy xuất bằng cách sử dụng chỉ mục âm: -1 đề cập đến mục cuối cùng, -2 đề cập đến mục cuối cùng thứ hai, v.v.

```
my_list = ["Hoc sinh", 1, True]
print(my_list[-1])
```



Truy xuất các phần tử vượt quá kích thước của List cũng gây ra lỗi:

- Trường hợp chỉ số index dương: **Index** \geq [Kích thước List]

```
my_list = ["Hoc sinh", 1, True]
print(my_list[3])
```

```
print(my_list[3])
IndexError: list index out of range
```

- Trường hợp chỉ số index âm: **Index** $<$ [Kích thước List] * (-1)

```
my_list = ["Hoc sinh", 1, True]
print(my_list[-4])
```

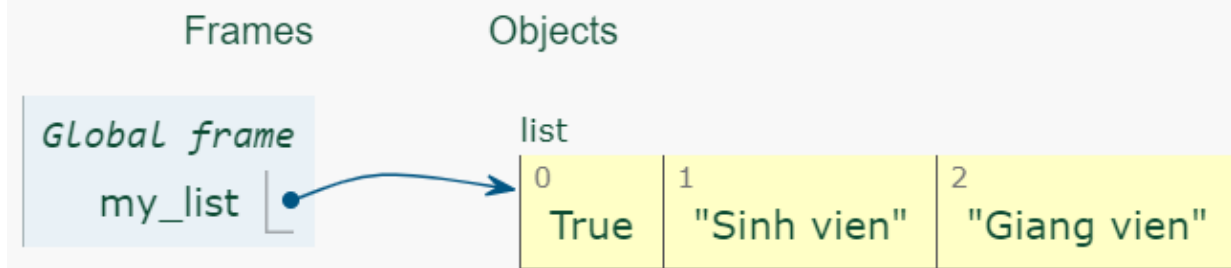
```
print(my_list[-4])
IndexError: list index out of range
```

1.2.4.2. Cập nhật List

Thay đổi giá trị của phần tử trong *list*:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
my_list[0] = True
print(my_list)
```

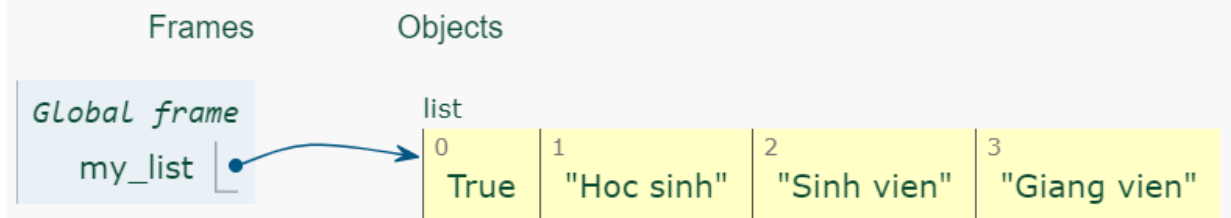
```
[True, 'Sinh vien', 'Giang vien']
```



*Thêm một phần tử mới vào **list**:*

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]  
my_list.insert(-5,True)  
print(my_list)
```

```
[True, 'Hoc sinh', 'Sinh vien', 'Giang vien']
```



Lưu ý: Đối với hàm **insert()** giá trị của index có thể vượt quá giới hạn của list.

Loại bỏ một phần tử ra khỏi list:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]  
my_list.remove("Sinh vien")  
print(my_list)
```

Loại bỏ phần tử bằng cách sử dụng chỉ mục:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
my_list.pop(0)
print(my_list)
```

Lưu ý: Nếu không chuyển đổi số chỉ mục cho phương thức **pop()**, thì phương thức này sẽ loại bỏ phần tử cuối cùng của **list**.

1.2.4.3. Vòng lặp với list

Chúng ta có thể lặp qua các phần tử của list bằng cách sử dụng vòng lặp for:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
for x in my_list:
    print(x)
```

Lặp qua các phần tử của list bằng cách sử dụng chỉ mục:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
for i in range(len(my_list)):
    print(my_list[i])
```

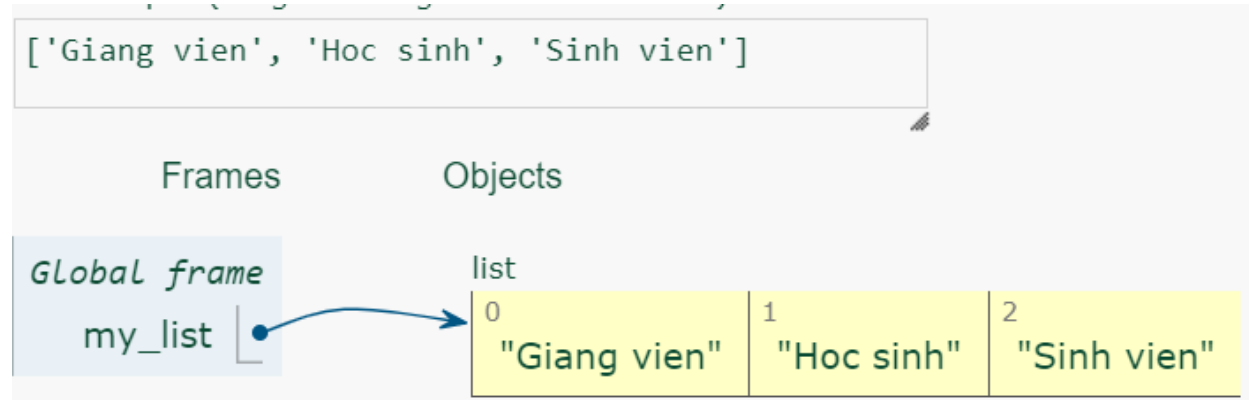
Lặp qua các phần tử của list bằng cách sử dụng vòng lặp while:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
i = 0
while i < len(my_list):
    print(my_list[i])
    i += 1
```

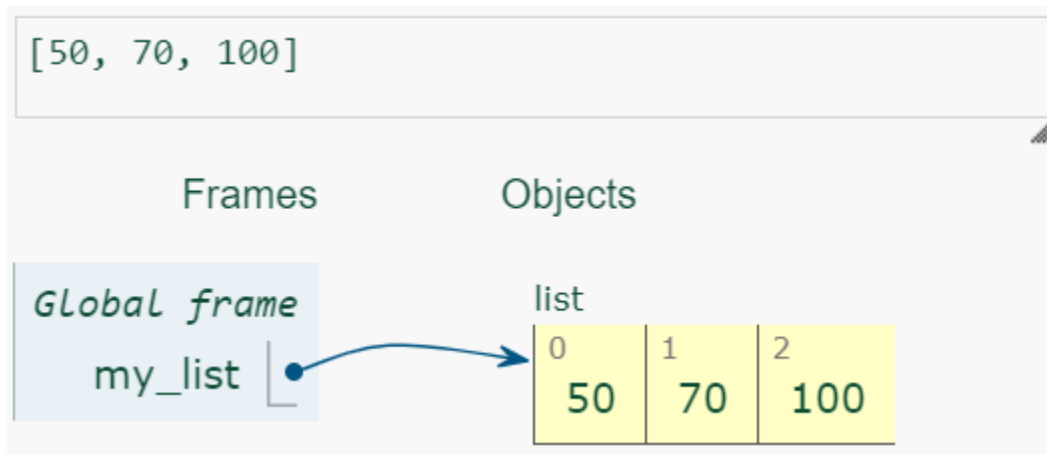
1.2.4.4. Sort list

Sắp xếp danh sách theo kiểu chữ và số:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
my_list.sort()
print(my_list)
```



```
my_list = [100, 50, 70]
my_list.sort()
print(my_list)
```



Lưu ý:

- Nếu muốn sort theo chiều giảm dần ta thêm đối số **reverse=True** vào hàm **sort()**
- Hàm **sort()** có phân chia theo chữ hoa và chữ thường, chữ hoa sẽ được ưu tiên trước

1.2.4.5. Bài tập

Câu hỏi

Chúng ta có một list như sau:

```
my_list = ["Hoc sinh", 1, False, "Sinh vien", 2, True]
```

- Bài tập 1: Điền cú pháp chính xác để in ra phần tử thứ hai của list
- Bài tập 2: Thay đổi giá trị “Sinh vien” thành “Giang vien”
- Bài tập 3: Dùng phương thức insert() để thêm vào “Giang vien” như là đối tượng thứ 2 trong list
- Bài tập 4: Sử dụng chỉ mục âm để in ra phần tử cuối cùng của list
- Bài tập 5: Điền cú pháp chính xác để in ra phần tử thứ 3, 4, 5

Đáp án

Bài tập 1:

```
print(my_list[1])
```

Bài tập 2:

```
for i in range(len(my_list)):
    if my_list[i] == "Sinh vien":
        my_list[i] = "Giang vien"
        break
print(my_list)
```

Bài tập 3:

```
my_list.insert(1, "Giang vien")
print(my_list)
```

Bài tập 4:

```
print(my_list[-1])
```

Bài tập 5:

```
print(my_list[2:5])
```

Lưu ý: Bài tập có thể có nhiều đáp án.

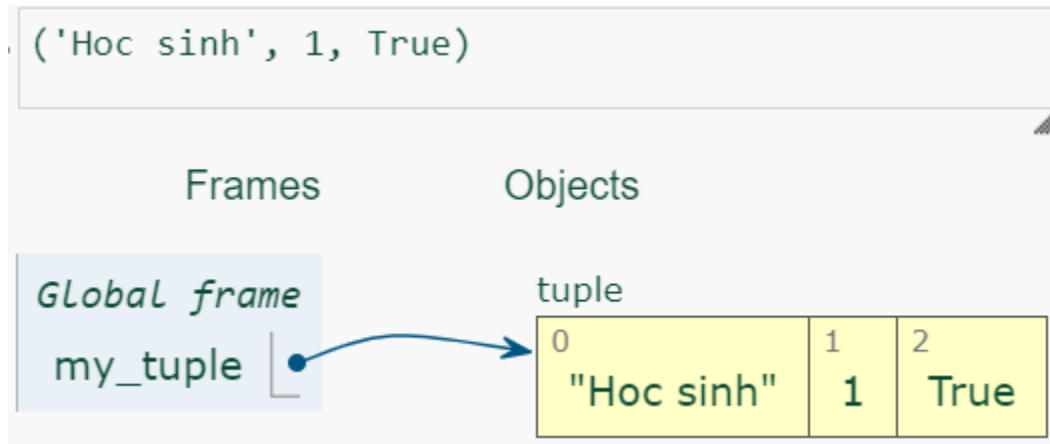
1.2.5. Tuple

Định nghĩa:

- Tuples được sử dụng để lưu trữ nhiều mục trong một biến duy nhất.
- Bộ tuple là một bộ sưu tập được sắp xếp theo thứ tự và không thể thay đổi.
- Tuples được viết bằng dấu ngoặc tròn.
- Các mục Tuple được sắp xếp theo thứ tự, có thể thay đổi và cho phép các giá trị trùng lặp. Tuple các mục được lập chỉ mục, mục đầu tiên có chỉ mục [0], mục thứ hai có chỉ mục [1], v.v.
- Tuple là không thể thay đổi, có nghĩa là chúng tôi không thể thay đổi, thêm hoặc xóa các mục sau khi bộ tuple đã được tạo.
- Cho phép các giá trị trùng lặp
- Các phần tử của Tuple có thể thuộc bất kỳ kiểu dữ liệu nào
- Để tạo một Tuple chỉ có một mục, chúng ta phải thêm dấu phẩy sau mục đó, nếu không Python sẽ không nhận ra đó là một Tuple.

Ví dụ:

```
my_tuple = ("Hoc sinh", 1, True)
print(my_tuple)
```

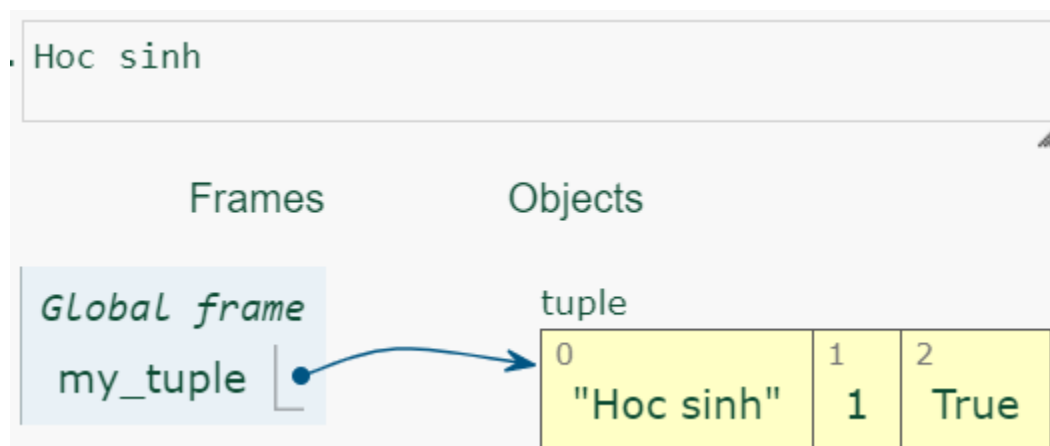
Độ dài của **Tuple** có thể được tính bằng cách sử dụng hàm **len()**:

```
my_tuple = ("Hoc sinh", 1, True)
print(len(my_tuple))
```

1.2.5.1. Truy xuất phần tử của Tuple

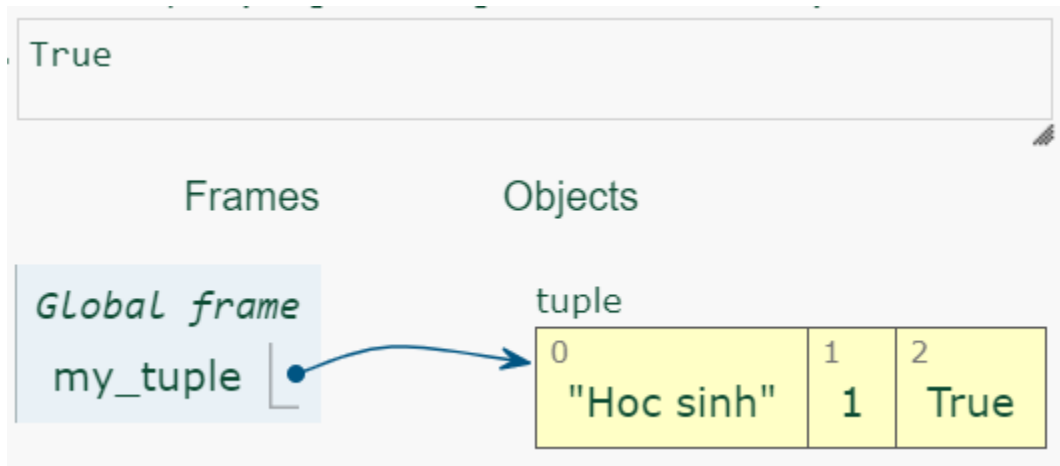
Chúng ta có thể truy cập nhiều mục bằng cách tham khảo số chỉ mục, bên trong dấu ngoặc vuông:

```
my_tuple = ("Hoc sinh", 1, True)
print(my_tuple[0])
```



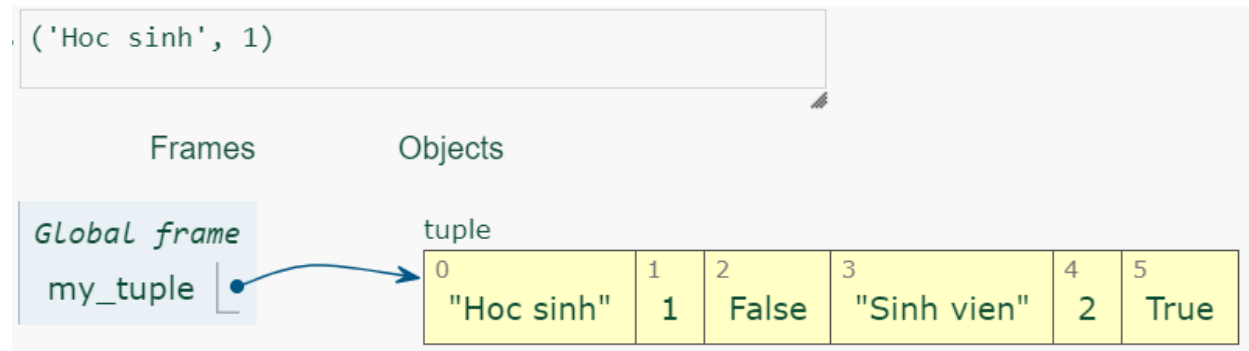
Truy xuất bằng cách sử dụng chỉ mục âm:

```
my_tuple = ("Hoc sinh", 1, True)
print(my_tuple[-1])
```

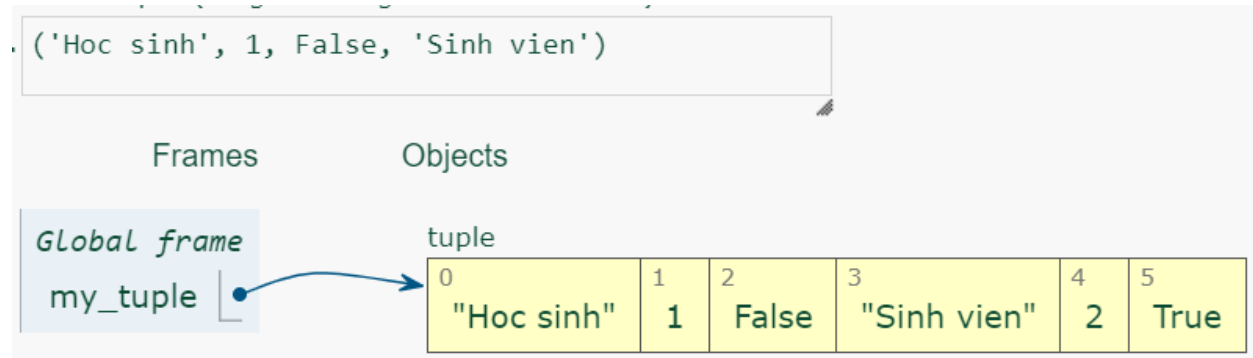


Truy xuất **Tuple** với khoảng các chỉ mục:

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
print(my_tuple[0:2])
```



```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
print(my_tuple[:4])
```



1.2.5.2. Cập nhật Tuple

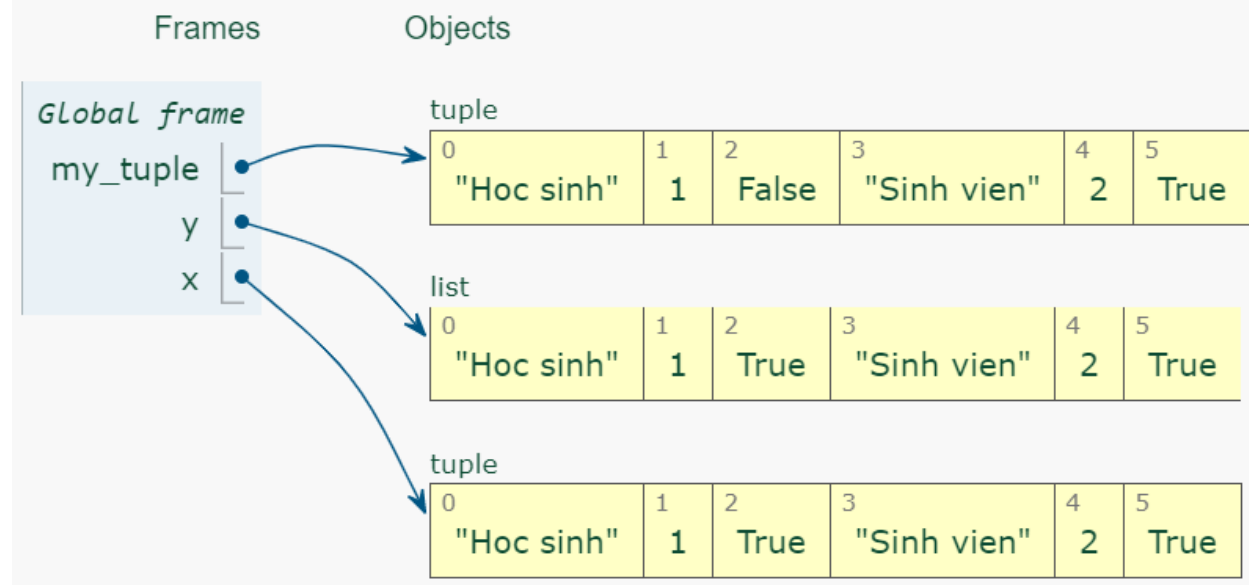
- Các **Tuple** không thể thay đổi, có nghĩa là ta không thể thay đổi, thêm hoặc xóa các phần tử trong tuple sau khi nó được tạo ra.
- Tuy nhiên có một số cách giải quyết vấn đề này.

*Thay đổi các giá trị của **Tuple**:*

- Chúng ta có thể chuyển đổi **Tuple** thành một danh sách, thay đổi danh sách và chuyển đổi lại danh sách thành một **Tuple**.

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
y = list(my_tuple)
y[2] = True
x = tuple(y)
print(x[:4])
```

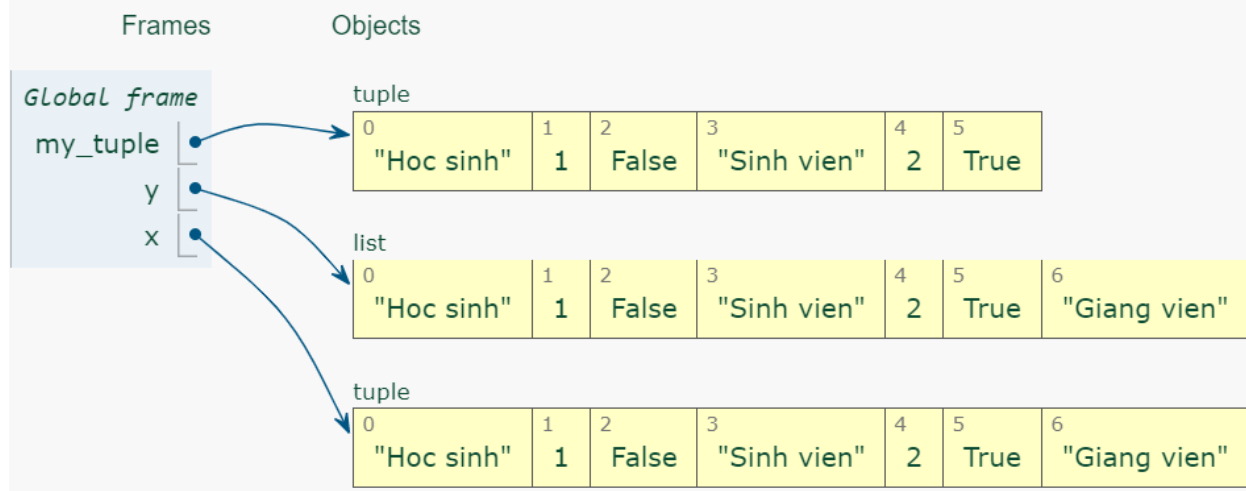
```
. ('Hoc sinh', 1, True, 'Sinh vien')
```



- Tương tự với việc thêm một phần tử mới vào **Tuple**:

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
y = list(my_tuple)
y.append("Giang vien")
x = tuple(y)
print(x)
```

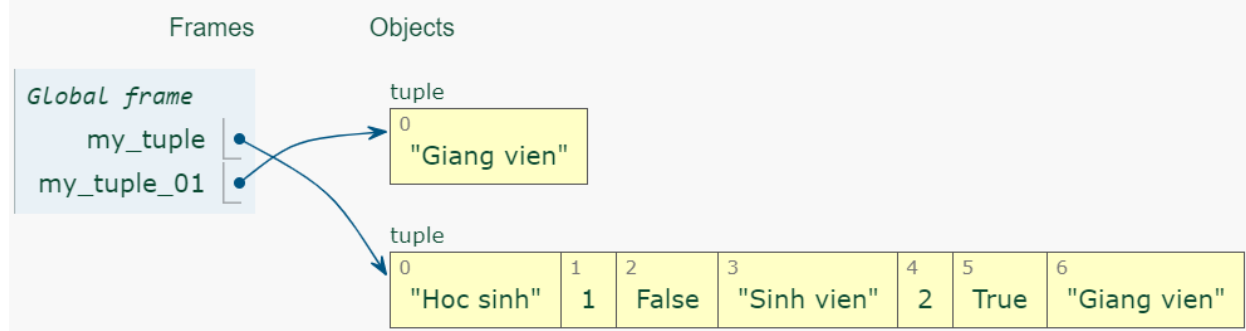
```
('Hoc sinh', 1, False, 'Sinh vien', 2, True, 'Giang vien')
```



- Thêm **Tuple** vào một **Tuple**

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
my_tuple_01 = ("Giang vien",)
my_tuple = my_tuple + my_tuple_01
print(my_tuple)
```

```
('Hoc sinh', 1, False, 'Sinh vien', 2, True, 'Giang vien')
```



1.2.5.3. Giải nén Tuple

- Khi tạo một Tuple, chúng ta thường gán các giá trị cho nó. Đây được gọi là “**packing**” một **Tuple**:

- Tuy nhiên, trong Python, chúng ta cũng được phép trích xuất các giá trị trở lại thành các biến. Đây được gọi là “**unpacking**”

Ví dụ:

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
(a,b,c,d,e,f) = my_tuple
print(a,b,c)
```

Print output (drag lower right corner to resize)

Hoc sinh 1 False

Frames

Objects

Global frame

my_tuple	
a	"Hoc sinh"
b	1
c	False
d	"Sinh vien"
e	2
f	True

tuple

0	1	2	3	4	5
"Hoc sinh"	1	False	"Sinh vien"	2	True

1.2.5.4. Vòng lặp với Tuple

- Lặp bằng cách sử dụng vòng **for**:

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
for x in my_tuple:
    print(x)
```

- Lặp bằng cách sử dụng chỉ số của Tuple:

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
for x in range(len(my_tuple)):
    print(x)
```

- Lặp bằng cách sử dụng vòng While: Sử dụng hàm **len()** để xác định độ dài của Tuple, sau đó bắt đầu từ 0 và lặp lại các mục Tuple bằng cách tham chiếu đến các chỉ mục của chúng. Đồng thời tăng chỉ số lên 1 sau mỗi lần lặp.

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
i = 0
while i < len(my_tuple):
    print(my_tuple[i])
    i = i + 1
```

1.2.5.5. Bài tập

Câu hỏi

Chúng ta có Tuple như sau:

```
my_tuple = ("Hoc sinh", 1, False, "Sinh vien", 2, True)
```

- Bài tập 1: Điền cú pháp chính xác để in ra số lượng phần tử của Tuple?
- Bài tập 2: Điền cú pháp chính xác để in ra phần tử đầu tiên của Tuple?
- Bài tập 3: Điền cú pháp chính xác để in ra phần tử cuối cùng của Tuple?
- Bài tập 4: Điền cú pháp chính xác để in ra phần tử thứ 3, 4, 5 của Tuple?

Đáp án

Bài tập 1:

```
print(len(my_tuple))
```

Bài tập 2:

```
print(my_tuple[0])  
# hoặc  
print(my_tuple[len(my_tuple)-1])
```

Bài tập 3:

```
print(my_tuple[len(my_tuple)-1])
```

Bài tập 4:

```
print(my_tuple[2:5])
```

1.2.6. Python Dictionary(Dict)

Định nghĩa:

- Dictionary được sử dụng để lưu trữ các giá trị dữ liệu trong các cặp key: value.
- Dictionary là một tập hợp được sắp xếp theo thứ tự *, có thể thay đổi và không cho phép trùng lặp.

Ví dụ:

```
my_dict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(my_dict)
```

Đặc điểm:

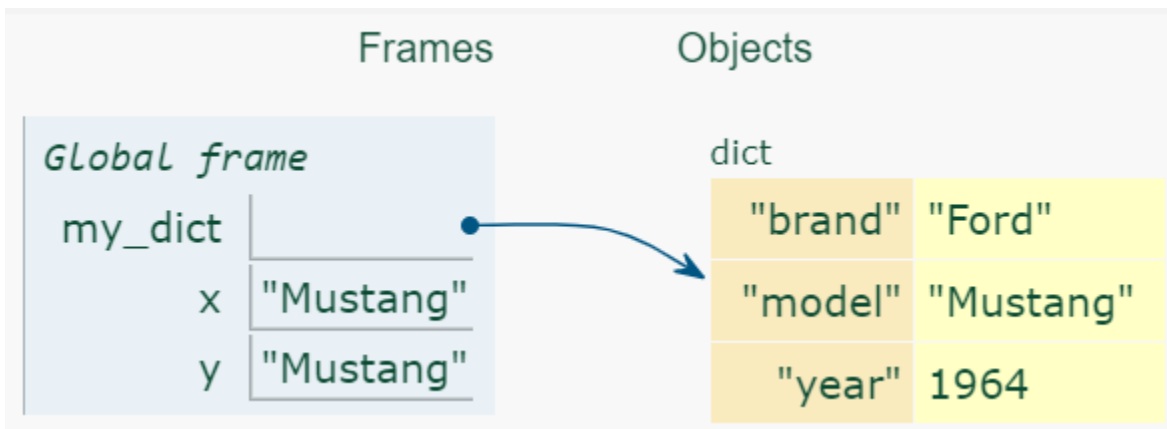
- Các mục của **dictionary** được sắp xếp theo thứ tự, có thể thay đổi và không cho phép trùng lặp.

- Các mục của **dictionary** được trình bày theo cặp **key: value** và có thể được tham chiếu bằng cách sử dụng tên khóa.
- **Dictionary** có thể thay đổi, nghĩa là chúng ta có thể thay đổi, thêm hoặc bớt các mục sau khi từ điển đã được tạo.
- **Dictionary** không thể có hai mục với cùng một khóa:

1.2.6.1. Truy xuất các phần tử của dictionary

Chúng ta có thể truy cập các mục của dictionary bằng cách tham khảo tên khóa bên trong dấu ngoặc vuông hoặc sử dụng get("key_name"):

```
my_dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = my_dict["model"]
y = my_dict.get("model")
```



Phương thức **Key()** sẽ trả về danh sách các key nằm trong dictionary. Ngoài ra, danh sách các key là một dạng view của dict nên bất kỳ thay đổi nào diễn ra với dict sẽ ảnh hưởng đến danh sách key:

```

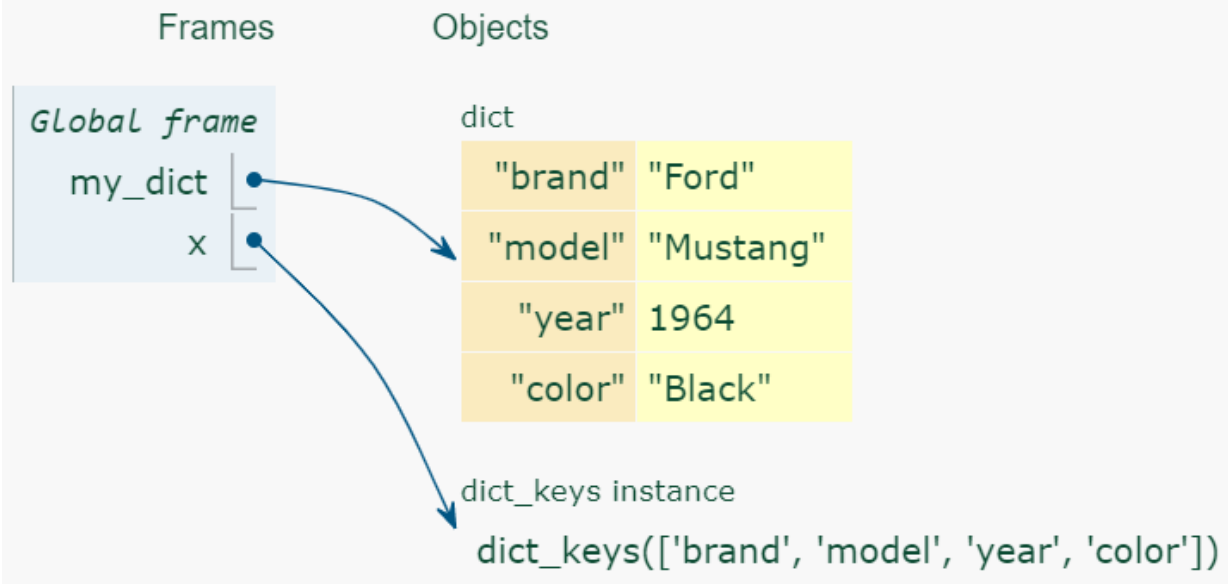
my_dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = my_dict.keys()
print(x)
my_dict["color"] = "Black"
print(x)

```

```

dict_keys(['brand', 'model', 'year'])
dict_keys(['brand', 'model', 'year', 'color'])

```



1.2.6.2. Cập nhật dict

Chúng ta có thể thay đổi giá trị của một phần tử nhất định trong dict bằng cách sử dụng key:

```
my_dict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
my_dict["model"] = "Yamaha"
```

Phương thức **update()** sẽ update những phần tử được cung cấp theo cặp dạng key : value. Một lưu ý rằng, các phần tử bên trong phải nằm bên trong một dict hoặc một phần tử có thể lặp:

```
my_dict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
my_dict.update({"model": "Yamaha", "year": "2021"})
```

Để thêm một phần tử mới vào dict, ta chỉ cần tạo một key mới và thêm value vào:

```
my_dict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
my_dict["color"] = "Blue"
```

Có 3 cách để xóa các phần tử ra khỏi dict:

- Sử dụng phương thức **pop("key")**

- Sử dụng phương thức **popItem()**, phương thức này sẽ loại bỏ phần tử cuối cùng của dict
- Sử dụng từ khóa **del**, nếu sử dụng từ khóa dict cho dict mà không cung cấp đối số key thì thay vào đó dict sẽ bị xóa hoàn toàn
- Sử dụng phương thức **clear()**, phương thức này sẽ làm rỗng dict

1.2.6.3. Vòng lặp với dict

Chúng ta có thể lặp qua các phần tử của dict bằng cách sử dụng vòng lặp for. Và các phần tử trả về sẽ là các key của dict:

```
my_dict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
for x in my_dict:
    print(x)
```

Để in ra giá trị của các value lần lượt, ta có thể làm như sau:

```
for x in my_dict:
    print(my_dict[x])
```

Ngoài ra ta cũng có thể sử dụng phương thức **values()** để đạt được kết quả như trên:

```
for x in my_dict.values():
    print(x)
```

Chúng ta có thể lặp đồng thời cả key và value thông qua phương thức **items()**

```
for x, y in my_dict.items():
    print(x, y)
```

1.2.6.4. Bài tập

Câu hỏi

Chúng ta dict như sau:

```
my_dict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

Bài tập 1: Thay đổi giá trị year từ 1964 sang 2021

Bài tập 2: Thêm cặp key : value color : red vào dict

Bài tập 3: Dùng pop để remove model ra khỏi dict

Bài tập 4: Làm trống dict

Đáp án

Bài tập 1:

```
my_dict["year"] = "2021"
```

Bài tập 2:

```
my_dict["color"] = "red"
```

Bài tập 3:

```
my_dict.pop("model")
```

Bài tập 4:

```
my_dict.clear()
```

1.3. Câu lệnh rẽ nhánh

Các loại câu lệnh điều kiện trong Python được mô tả trong bảng dưới:

Equals	<code>a == b</code>
Not Equals	<code>a != b</code>
Less than	<code>a < b</code>
Less than or equal to	<code>a <= b</code>
Greater than	<code>a > b</code>
Greater than or equal to	<code>a >= b</code>

Ví dụ:

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

1.4. Vòng lặp

1.4.1. Vòng lặp For

Vòng lặp for được sử dụng để lặp qua một đối tượng có sự liên tiếp (có thể là một list, tuple, dictionary, một tập hợp hoặc một chuỗi).

Ví dụ:

```
my_list = ["Hoc sinh", "Sinh vien", "Giang vien"]
for x in my_list:
    print(x)
```

Để thoát vòng lặp ngay lập tức ta có thể sử dụng lệnh **break**.

1.4.2. Vòng lặp while

Với vòng lặp while, chúng ta có thể thực hiện một tập hợp các câu lệnh miễn là một điều kiện là đúng.

Ví dụ:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

1.5. Hàm trong Python

Định nghĩa:

- Hàm là một khối mã chỉ chạy khi nó được gọi.
- Chúng ta có thể truyền dữ liệu, được gọi là tham số, vào một hàm.
- Một hàm có thể trả về dữ liệu như là kết quả.

Để tạo một hàm ta sử dụng từ khóa **def**:

```
def my_function():
    print("Hello from a function")
```

Để gọi một hàm, hãy sử dụng tên hàm theo sau bởi dấu ngoặc đơn:

```
def my_function():
    print("Hello from a function")
my_function()
```

Arguments (Đối số):

- Dữ liệu có thể được chuyển vào các hàm dưới dạng đối số.
- Các đối số được chỉ định sau tên hàm, bên trong dấu ngoặc đơn. Ta có thể thêm bao nhiêu đối số tùy thích, chỉ cần phân tách chúng bằng dấu phẩy.

Ví dụ:

```
def my_function(fname):
    print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Lưu ý: Khi gọi hàm chúng ta cần truyền vào đầy đủ các đối số của hàm.

***kwargs: Được sử dụng khi chúng ta không biết số lượng chính xác các đối số cần thiết để truyền vào hàm. Bằng cách này hàm sẽ nhận vào một dictionary của các đối số, Ví dụ:*

```
def my_function(**kid):
    print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "fate")
```

Return: Để trả về giá trị từ hàm ta sử dụng từ khóa return, ví dụ:

```
def my_function(x):
    return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Python có hỗ trợ đệ quy nên chúng ta hoàn toàn có thể gọi hàm trong hàm, ví dụ:


```
def tri_recursion(k):
    if(k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

6. Lớp và đối tượng trong Python

Định nghĩa:

- Python là một ngôn ngữ lập trình hướng đối tượng.
- Hầu hết mọi thứ trong Python đều là một đối tượng, với các thuộc tính và phương thức.
- Một Lớp giống như một phương thức khởi tạo đối tượng

Tạo một class:

Để tạo một class ta sử dụng từ khóa class:

```
class MyClass:
    x = 5
```

Sau khi tạo class, chúng ta có thể khởi tạo một đối tượng thuộc lớp này:

```
p1 = MyClass()
print(p1.x)
```

Hàm __init__():

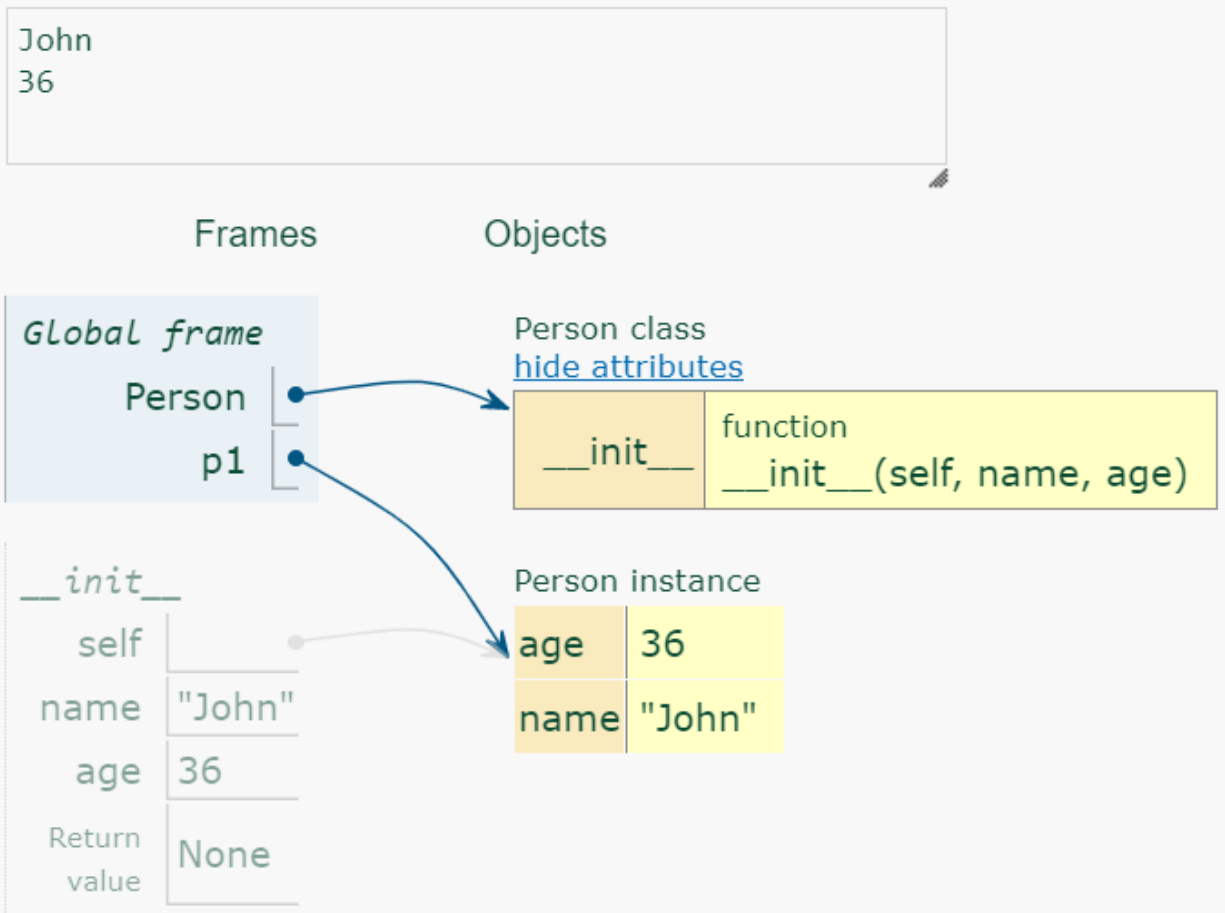
- Tất cả các lớp đều có một hàm được gọi là `__init__()`, hàm này luôn được thực thi khi lớp đang được khởi tạo.
- Sử dụng hàm `__init__()` để gán giá trị cho thuộc tính đối tượng hoặc các thao tác khác cần thực hiện khi đối tượng đang được tạo:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

```
p1 = Person("John", 36)
```

```
print(p1.name)
```

```
print(p1.age)
```



Phương thức của đối tượng là một hàm thuộc Lớp, ví dụ:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Tham số self:

- Tham số self là một tham chiếu đến thể hiện hiện tại của lớp và được sử dụng để truy cập các biến thuộc về lớp.
- Nó không nhất thiết phải được đặt tên là self, tuy nhiên tham số này luôn luôn phải được đặt ở đầu tiên trong danh sách các tham số.

Ví dụ:

```
class Person:
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age

    def myfunc(abc):
        print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

1.6. Kế thừa trong Python

Tính kế thừa cho phép chúng ta định nghĩa một lớp kế thừa tất cả các phương thức và thuộc tính từ một lớp khác.

- Lớp cha là lớp được kế thừa, còn được gọi là lớp cơ sở.
- Lớp con là lớp kế thừa từ lớp khác, còn được gọi là lớp dẫn xuất.

Tạo một Lớp cha:

```

class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
# Tạo Lớp Person và thực thi phương thức printname
x = Person("John", "Doe")
x.printname()

```

Để tạo một lớp kế thừa chức năng từ một lớp khác, hãy gửi lớp cha làm tham số khi tạo lớp con:

```

class Student(Person):
    pass

x = Student("NAM", "LAN")
x.printname()

```

Hàm __init__():

Nếu chúng ta thêm vào hàm __init__() hàm con sẽ không còn kế thừa từ lớp cha.

Để giữ lại các kế thừa từ lớp cha, ta gọi lớp cha trong hàm __init__():

```

class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)

```

Tương tự ta cũng có thể sử dụng hàm **super()**:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(self, fname, lname)
```

Để thêm thuộc tính vào lớp kết thừa ta làm như sau:

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
```

CHƯƠNG 2: DJANGO

2.1. Django

Django là một web framework miễn phí, mã nguồn mở dựa trên Python và hỗ trợ các mẫu kiến trúc model-template-view. Framework này giúp việc phát triển ứng dụng web một cách nhanh chóng và hiệu quả. Đồng thời Django cũng giải quyết các vấn đề phức tạp của việc phát triển web. Ví dụ: kết nối giữa cơ sở dữ liệu và moodle, Django admin với giao diện trực quan, tự động cài đặt, cho phép tùy chỉnh,...

Các tính năng nổi bật của Django framework:

- **Nhanh:** Django được thiết kế để giúp các nhà phát triển tạo các ứng dụng web rất nhanh từ khi lên ý tưởng cho đến khi hoàn thành.
- **Bảo mật:** Django có nhiều thư viện phần mềm trung gian được tích hợp sẵn để xử lý các lỗi bảo mật phổ biến mà các nhà phát triển mắc phải, chẳng hạn như SQL injection, cross-site scripting hoặc CSRF attack.
- **Khả năng mở rộng:** Nhiều trang web có lưu lượng truy cập cao được phát triển bằng Django (youtube, Instagram v.v). Django có khả năng mở rộng quy mô trang web một cách nhanh chóng và linh hoạt để đáp ứng nhu cầu lưu lượng truy cập lớn nhất.

- **“Batteries included”**: Django framework bao gồm nhiều thư viện có thể được sử dụng để xử lý các tác vụ phát triển web thông thường, như xác thực người dùng, quản lý nội dung, nguồn cấp dữ liệu RSS, v.v.
- **Đa năng**: Django có thể được sử dụng cho hầu hết các loại website, từ hệ thống quản lý nội dung đến mạng xã hội, blog,... Từ các website nhỏ đến các website khổng lồ.

1.1. Cài đặt Django

2.1.1. Thiết lập môi trường ảo

Đối với mỗi dự án web, nên có một môi trường ảo riêng dành cho project đó. Vì các gói thư viện cài đặt vào mỗi dự án web sẽ khác nhau.

Chọn thư mục sử dụng cho Django project và tiến hành thiết lập môi trường ảo (virtualenv).

Để tạo môi trường ảo ta chạy đoạn command sau trên command line:

```
python3 -m venv <ten_moi_truong_ao>-env
```

Sau khi hoàn thành việc tạo môi trường ảo, ta tiến hành việc kích hoạt môi trường ảo:

Window:

```
<ten_moi_truong_ao>-env\Scripts\activate.bat
```

Unix, Mac/Os:

```
source <ten_moi_truong_ao>-env/bin/activate
```

2.1.1.1. Cài đặt pip

Sau khi hoàn thành việc kích hoạt môi trường ảo ta thực hiện việc cài đặt **pip** để cài đặt các python package, cũng như update và remove package

Window

```
py -m ensurepip --upgrade
```

Linux, MacOS

```
python -m ensurepip --upgrade
```

2.1.2. Cài đặt cơ sở dữ liệu

2.1.2.1. PostgreSQL

PostgreSQL là một hệ cơ sở dữ liệu mã nguồn mở mạnh mẽ, sử dụng và mở rộng từ ngôn ngữ SQL kết hợp với nhiều tính năng khác. PostgreSQL nổi tiếng nhờ kiến trúc đã được chứng minh bởi độ tin cậy, tính toàn vẹn dữ liệu, bộ tính năng mạnh mẽ, khả năng mở rộng và sự cống hiến của cộng đồng mã nguồn mở đằng sau phần mềm để liên tục cung cấp các giải pháp hiệu quả và sáng tạo. PostgreSQL có thể chạy trên tất cả các hệ điều hành.

Mô hình cấp phép mã nguồn mở cũng ít tốn kém hơn nhiều so với Oracle hoặc các cơ sở dữ liệu độc quyền khác.

Cộng đồng PostgreSQL cũng thường xuyên phát hành các bản cập nhật với các tính năng tuyệt vời. Ví dụ: trong bản phát hành PostgreSQL 9.4, kiểu JSONB đã được thêm vào để nâng cao khả năng lưu trữ JSON, bản phát hành PostgreSQL 9.6 đã thêm tìm kiếm cụm từ, bản phát hành PostgreSQL 10 đã thêm kiểu dữ liệu macaddr8, v.v.

Một số tính năng chính của PostgreSQL:

- User-defined datatypes
- Table inheritance
- Foreign key referential integrity
- Views, rules
- Triggers
- Streaming replication
- Hot standby
- Nested transactions
- Tablespaces
- Point-in-time recovery (PITR)
- Hỗ trợ truy vấn SQL (quan hệ) và JSON (không quan hệ)
- Tuân thủ tiêu chuẩn ANSI SQL

2.1.2.2. Lợi ích của việc sử dụng PostgreSQL với Django

Sử dụng PostgreSQL và Django cùng nhau mang lại nhiều lợi ích:

- Django cung cấp một số kiểu dữ liệu sẽ chỉ hoạt động với PostgreSQL.
- Django có `django.contrib.postgres` để thực hiện các hoạt động cơ sở dữ liệu trên PostgreSQL.
- Nếu chúng ta đang xây dựng một ứng dụng với bản đồ hoặc đang lưu trữ dữ liệu địa lý, ta cần sử dụng PostgreSQL, vì GeoDjango chỉ hoàn toàn tương thích với PostgreSQL.
- PostgreSQL có bộ tính năng phong phú nhất được hỗ trợ bởi Django.

Dưới đây là một số tính năng dành riêng cho PostgreSQL được Django hỗ trợ:

- PostgreSQL-specific aggregation functions
- PostgreSQL-specific database constraints
- PostgreSQL-specific form fields and widgets
- PostgreSQL-specific database functions
- PostgreSQL-specific model indexes
- PostgreSQL-specific lookups
- Database migration operations
- Full-text search
- Validators

So với PostgreSQL, MySQL thiếu hỗ trợ cho các transaction xung quanh các hoạt động thay đổi lược đồ, có nghĩa là nếu quá trình migration không áp dụng được, chúng ta sẽ phải bỏ chọn các thay đổi theo cách thủ công sau đó thử migration lại (không thể quay lại điểm trước đó).

2.1.2.3. Cài đặt PostgreSQL

Để cài đặt PostgreSQL, tải bản cài đặt tại trang chủ của PostgreSQL:

<https://www.postgresql.org/download/>

Chạy file setup, tiến hành cài đặt.

2.1.3. Tiến hành cài đặt Django

Trước khi cài đặt Django sau khi kích hoạt môi trường ảo (virtualenv) ta cần cài đặt package cần thiết cho PostgreSQL:

```
pip install psycopg2
```

Để cài đặt Django ta chạy đoạn lệnh sau trên command line:

```
pip install Django
```

Sau khi cài đặt thành công, kiểm tra command django-admin:

```
Type 'django-admin help <subcommand>' for help on a specific subcommand.
```

```
Available subcommands:
```

```
[django]
  check
  compilemessages
  createcachetable
  dbshell
  startapp
  startproject
  test
  testserver
```

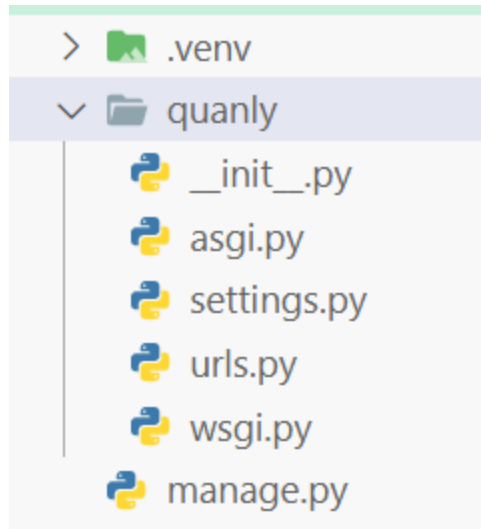
2.1.4. Khởi tạo dự án web đầu tiên

Để tạo project, ta sử dụng command sau:

```
django-admin startproject project-name .
```

Lưu ý: Nếu muốn tạo project kèm với thư mục root chứa project thì ta bỏ đi dấu “.” của cuối đoạn lệnh.

Sau khi cài đặt thành công, cây thư mục sẽ có dạng:



Chạy lệnh “**python manage.py migrate**” để áp dụng những thay đổi cho admin, session v.v mặc định của Django:

Operations to perform:

Apply all migrations: admin, auth, contenttypes, sessions

Running migrations:

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying sessions.0001_initial... OK
```

Chạy lệnh “**py manage.py runserver**” để kiểm tra việc cài đặt hoàn tất:



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.

2.1.4.1. Tạo ứng dụng đầu tiên

Để tạo một ứng dụng trong Django project ta sử dụng command sau:

```
django-admin startapp myapp
```

```
> .venv
✓ myapp
  > migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  > quanly
    db.sqlite3
    des.txt
    manage.py
```

Để Django nhận diện ứng dụng mới được cài đặt và chúng ta muốn sử dụng nó, ta thêm **<app_name>** vào tệp **settings.py** trong thư mục của **project** tại phần **INSTALLED_APPS**:

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

2.1.4.2. Cài đặt cấu hình Databases

Theo mặc định Django cấu hình sử dụng SQLite như sau:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': BASE_DIR / 'db.sqlite3',  
    }  
}
```

Chúng ta có thể thay đổi cài đặt ở đây để sử dụng database mà chúng ta mong muốn, kể cả local hay remote database. Giả sử trong trường hợp chúng ta sử dụng PostgreSQL, ta có thể cấu hình như sau (các thông số của database có trong lúc cài đặt hoặc sử dụng pgAdmin để lấy các thông số database cần thiết):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'database_name',
        'USER': 'postgres',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Và thế là tất cả mọi thứ đã sẵn sàng để chúng ta tiếp bước trên con đường chinh phục Django.

2.2. Models và Databases

2.2.1. Models

Models là phần định nghĩa mô hình dữ liệu của Django. Nó chứa các trường (field) và hành vi (behaviour) thiết yếu của dữ liệu đang lưu trữ trong database. Mỗi một model sẽ được tham chiếu với một bảng nằm trong database.

Về mặt cơ bản:

- Mỗi Model là một lớp của Python thuộc phân lớp `django.db.models.Model`.
- Mỗi thuộc tính của model đại diện cho một trường cơ sở dữ liệu.
- Dựa vào model, Django cung cấp một automatically-generated database-access API.

Ví dụ:

- Ví dụ này định nghĩa model `SinhVien` với các thuộc tính như **mssv** (Mã số sinh viên), **ho_va_ten**.

```
from django.db import models

# Create your models here.
class SinhVien(models.Model):
    mssv = models.CharField(default="", max_length=8)
    ho_va_ten = models.CharField(default="", max_length=25)
```

- **mssv** và **ho_va_ten** là các trường của model. Mỗi trường được chỉ định là một thuộc tính của lớp và mỗi thuộc tính ánh xạ tới một cột cơ sở dữ liệu.
- Model SinhVien ở trên sẽ tạo một bảng cơ sở dữ liệu như sau:

```
CREATE TABLE quanly_sinhvien (
    "id" serial NOT NULL PRIMARY KEY,
    "mssv" varchar(8) NOT NULL,
    "ho_va_ten" varchar(25) NOT NULL
);
```

- Một số lưu ý:
 - Tên bảng **quanly_sinhvien** được tự động tạo dựa trên tên app kết hợp với tên bảng và có thể bị ghi đè
 - Trường **id** được tự động tạo. Tuy nhiên có thể bị ghi đè
 - **SQL CREATE TABLE** trong ví dụ này được định dạng bằng cú pháp PostgreSQL và có thể bị ghi đè trong tập tin **settings.py**

Sử dụng model:

- Sau khi định nghĩa các model, điều quan trọng cần làm tiếp theo đó chính là làm cho Django biết việc chúng ta sẽ sử dụng những model này. Bằng cách sửa đổi file **settings.py** tại mục **INSTALLED_APPS** thành tên của module chứa **models.py**
- Ví dụ: Nếu **models.py** nằm bên trong module **quanly.models** thì trong **INSTALLED_APPS** ta thêm vào dòng sau:

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

- Sau khi hoàn thành việc thêm vào **INSTALLED_APPS**, chúng ta tiến hành chạy lệnh **manage.py makemigrations** để tiến hành các thay đổi, rồi tiếp đến chạy lệnh **manage.py migrate** để áp dụng các thay đổi

2.2.1.1. Fields (Các trường)

Thành phần quan trọng nhất của một model – và là thành phần bắt buộc duy nhất của model – là danh sách các trường cơ sở dữ liệu mà nó xác định. Các trường được chỉ định bởi các thuộc tính của lớp. Cần thận khi đặt tên trường trùng với các model API như **clean**, **save** hoặc **delete**.

Các loại trường:

Mỗi trường trong model phải là một thể hiện của field Class thích hợp. Django sử dụng các loại field Class để xác định:

- Loại cột, cho cơ sở dữ liệu biết loại dữ liệu nào cần lưu trữ (ví dụ: INTEGER, VARCHAR, TEXT).
- Tiện ích HTML mặc định để sử dụng để render một trường form (ví dụ: <input type = "text">, <select>).
- Các yêu cầu xác thực tối thiểu, được sử dụng cho Django admin và trong các form được tạo tự động.

Các tùy chọn:

- Mỗi trường có một tập hợp các đối số cụ thể. Ví dụ: **CharField** (và các lớp con của nó) yêu cầu đối số `max_length` chỉ định kích thước của trường **VARCHAR** trong cơ sở dữ liệu.
- Ngoài ra còn có một tập hợp các đối số chung cho tất cả các loại trường. Tất cả đều là tùy chọn.
 - **null:** Nếu True, Django sẽ lưu trữ các giá trị trống dưới dạng NULL trong cơ sở dữ liệu. Mặc định là False.
 - **blank:**
 - Nếu True, trường được phép để trống. Mặc định là False.
 - Lưu ý: blank khác với null ở chỗ, null liên quan đến việc lưu trữ trong cơ sở dữ liệu còn blank liên quan đến việc xác thực các trường bị bỏ trống khi sử dụng Django form
 - **choices:** Một chuỗi 2-tuple được sử dụng làm lựa chọn cho trường. Nếu được lựa chọn, form widget mặc định sẽ là một select box thay vì text field chuẩn và sẽ giới hạn các lựa chọn dựa trên các lựa chọn đã cho.

```
PHAN_LOAI = [  
    ('XS', 'Xuất sắc'),  
    ('G', 'Giỏi'),  
    ('K', 'Khá'),  
    ('TB', 'Trung bình'),  
    ('Y', 'Yếu'),  
]
```

- Phần tử đầu tiên trong mỗi tuple là giá trị sẽ được lưu trữ trong cơ sở dữ liệu. Phần tử thứ hai được hiển thị bởi form widget.

```
class SinhVien(models.Model):
    PHAN_LOAI = [
        ('XS', 'Xuat sac'),
        ('G', 'Gioi'),
        ('K', 'Kha'),
        ('TB', 'Trung binh'),
        ('Y', 'Yeu'),
    ]
    mssv = models.CharField(default="", max_length=8)
    ho_va_ten = models.CharField(default="", max_length=25)
    loai = models.CharField(max_length=1, choices=PHAN_LOAI)
```

- default: Giá trị mặc định cho trường. Được gọi bất cứ khi nào đối tượng được khởi tạo.
- **primary_key:**
 - Nếu True, trường này được sử dụng làm khóa chính
 - Nếu chúng ta không chỉ định **primary_key = True** cho bất kỳ trường nào trong model, Django sẽ tự động thêm **IntegerField** để làm khóa chính, vì vậy ta không cần đặt **primary_key = True** trên bất kỳ trường nào trừ khi ta muốn ghi đè hành vi khóa chính mặc định.
- **unique:**
 - Nếu True, trường này phải là duy nhất trong toàn bộ bảng.

2.2.1.2. Các mối quan hệ (Relationships)

Điểm mạnh của cơ sở dữ liệu quan hệ là các bảng liên quan đến nhau. Django cung cấp 3 cách để định nghĩa những loại quan hệ cơ bản nhất của hệ cơ sở dữ liệu quan hệ: many-to-one, many-to-many and one-to-one.

Many-to-one:

- Để xác định mối quan hệ *Many-to-one* ta sử dụng **django.db.models.ForeignKey**. Trường này được sử dụng giống như những trường khác của model bằng cách đưa nó làm lớp thuộc tính của model.

- **ForeignKey** yêu cầu một đối số tham chiếu đến model liên quan.
- Ví dụ: Nếu mỗi **Sinh Viên** thuộc một **Niên Khóa**. Với mỗi niên khóa sẽ có nhiều sinh viên nhưng mỗi sinh viên chỉ thuộc một niên khóa. Chúng ta có thể biểu diễn mối quan hệ đó như sau:

```
class NienKhoa(models.Model):
    khoa = models.IntegerField()
    nam = models.IntegerField()

class SinhVien(models.Model):
    mssv = models.CharField(default="", max_length=8)
    ho_va_ten = models.CharField(default="", max_length=25)
    khoa = models.ForeignKey(NienKhoa, on_delete=models.CASCADE)
```

- Chúng ta đồng thời có thể tạo nên mối quan hệ **many-to-one** với chính nó. Ví dụ: Một môn học có thể có một môn học tiên quyết khác và đồng thời một môn học có thể là môn học tiên quyết cho nhiều môn học khác.

```
class MonHoc(models.Model):
    ten_mon_hoc = models.CharField(max_length=25)
    mon_hoc_tien_quyet = models.ForeignKey("self", on_delete=models.CASCADE)
```

Many-to-many:

- Để xác định mối quan hệ *Many-to-many*, chúng ta sử dụng **ManyToManyField**.
- **ManyToManyField** yêu cầu một đối số tham chiếu đến model liên quan
- Ví dụ: Một môn học có thể có nhiều sinh viên tham gia học và ngược lại một sinh viên có thể học nhiều môn học.

```
class MonHoc(models.Model):
    ten_mon_hoc = models.CharField(max_length=25)
    mon_hoc_tien_quyet = models.ForeignKey("self", on_delete=models.CASCADE)

class SinhVien(models.Model):
    mssv = models.CharField(default="", max_length=8)
    ho_va_ten = models.CharField(default="", max_length=25)
    khoa = models.ForeignKey(NienKhoa, on_delete=models.CASCADE)
    mon_hoc = models.ManyToManyField(MonHoc)
```

One-to-one:

Để xác định mối quan hệ One-to-one, chúng ta sử dụng **OneToOneField**

OneToOneField yêu cầu một đối số tham chiếu đến model liên quan

Ví dụ: Một sinh viên chỉ có một hồ sơ duy nhất và một hồ sơ cũng chỉ thuộc một sinh viên duy nhất.

```
class HoSo(models.Model):
    dia_chi = models.CharField(max_length=500)
    ngay_cap_nhat = models.DateTimeField(auto_now_add=True)

class SinhVien(models.Model):
    mssv = models.CharField(default="", max_length=8)
    ho_va_ten = models.CharField(default="", max_length=25)
```

Tùy chọn meta:

Model metadata là “bất kỳ thứ gì không phải là trường”, chẳng hạn như tùy chọn sắp xếp (order), tên bảng cơ sở dữ liệu (db_table) hoặc tên số nhiều và số ít mà con người có thể đọc được (verbose_name và verbose_name_plural). Không bắt buộc phải có, và việc thêm Class Meta vào một model là hoàn toàn tùy chọn. Ví dụ:

```
class SinhVien(models.Model):
    mssv = models.CharField(default="", max_length=8)
    ho_va_ten = models.CharField(default="", max_length=25)

    class Meta:
        ordering = ["id"]
```

2.2.2. Tạo câu truy vấn

Sau khi đã hoàn thành việc tạo các model liệu, Django sẽ tự động cung cấp một API trừu tượng hóa cơ sở dữ liệu cho phép chúng ta tạo, truy xuất, cập nhật và xóa các đối tượng.

Tạo đối tượng (objects):

- Để tạo một đối tượng, hãy khởi tạo đối tượng đó bằng cách sử dụng từ khóa cho model class, sau đó gọi `save()` để lưu vào cơ sở dữ liệu.
- Giả sử các mô hình nằm trong tệp `quanly / models.py`, ví dụ:

```
from quanly.models import SinhVien
s = SinhVien(ho_va_ten='Tran Van A', mssv="17731425")
s.save()
```

- Điều này thực hiện một câu lệnh **INSERT SQL**. Django chỉ tiến hành gọi database để thực hiện câu truy vấn khi chúng ta gọi phương thức `save()`.
- Phương thức `save()` không có giá trị trả về.

Lưu thay đổi đối với các đối tượng:

- Để lưu các thay đổi đối với một đối tượng đã có trong cơ sở dữ liệu, ta sử dụng phương thức `save()`.

```
s.ho_va_ten = "Tran Van B"
s.save()
```

- Điều này thực hiện một câu lệnh **UPDATE SQL**. Django chỉ tiến hành gọi database để thực hiện câu truy vấn khi chúng ta gọi phương thức **save()**.

Lưu thay đổi đối với các trường ForeignKey và ManyToManyField

Cập nhật trường **ForeignKey** hoạt động tương tự như cách lưu các trường bình thường khác. Ví dụ này cập nhật môn học tiên quyết của một môn học (Giả sử các môn học đã được lưu vào cơ sở dữ liệu từ trước đó):

```
mon_hoc = MonHoc.objects.get(pk=1)
mh_tien_quyet = MonHoc.objects.get(pk=2)
mon_hoc.mon_hoc_tien_quyet = mh_tien_quyet
mon_hoc.save()
```

UPDATE ManyToManyField hoạt động hơi khác một chút - sử dụng phương thức **add()** của trường để thêm bản ghi vào quan hệ. Ví dụ sau thêm đối tượng Môn học vào đối tượng Thời khóa biểu:

```
tkb = ThoiKhoaBieu.objects.get(pk=1)
mh = MonHoc.objects.get(pk=1)
tkb.mon_hoc.add(mh)
tkb.save()
```

Truy xuất đối tượng

Truy xuất toàn bộ:

Cách đơn giản nhất để lấy các đối tượng từ một bảng là lấy tất cả. Để thực hiện chúng ta sử dụng phương thức **all()**

```
ds_sv = SinhVien.objects.all()
```

Truy xuất đối tượng bằng bộ lọc (filters):

QuerySet được trả về bởi `all()` mô tả tất cả các đối tượng trong bảng cơ sở dữ liệu. Tuy nhiên, thông thường, chúng ta sẽ chỉ cần chọn một tập hợp con của tập hợp hoàn chỉnh của tập các đối tượng.

Để tạo một tập hợp con như vậy, ta cần tinh chỉnh QuerySet ban đầu, thêm các điều kiện lọc. Hai cách phổ biến nhất để tinh chỉnh QuerySet là:

- **`filter(**kwargs)`**: Trả về một QuerySet mới chứa các đối tượng phù hợp với các tham số tra cứu đã cho.
- **`exclude(**kwargs)`**: Trả về một QuerySet mới chứa các đối tượng không khớp với các tham số tra cứu đã cho.

Các tham số (`**kwargs` trong định nghĩa hàm ở trên) nên ở định dạng được mô tả trong query bên dưới:

```
s = SinhVien.objects.filter(ho_va_ten="Tran Van A")
```

Bộ lọc chuỗi (Chaining filters):

- Chúng ta có thể kết hợp liên tục các bộ lọc để thu được queryset cuối cùng

```
s = SinhVien.objects.filter(ho_va_ten="Tran Van A").exclude(mssv="12345678")
```

Truy xuất một đối tượng duy nhất bằng `get()`

- **`filter()`** sẽ luôn trả về cho chúng ta một QuerySet, ngay cả khi chỉ một đối tượng duy nhất phù hợp với truy vấn - trong trường hợp này, nó sẽ là một QuerySet chứa một phần tử duy nhất.
- Nếu chúng ta biết chỉ có một đối tượng phù hợp với truy vấn của mình, ta có thể sử dụng phương thức **`get()`** để trả về trực tiếp đối tượng:

```
tkb = ThoiKhoaBieu.objects.get(pk=1)
```

Lưu ý: Có sự khác biệt giữa **get()** và **filter()** khi đối tượng truy vấn không tồn tại trong cơ sở dữ liệu. Trường hợp **get()**, nếu không có đối tượng phù hợp với câu truy vấn thì **DoesNotExist exception** sẽ được thông báo. Còn đối với **filter()** ở trường hợp tương tự, sẽ trả một queryset rỗng.

Giới hạn QuerySets

- Sử dụng kỹ thuật cắt mảng của Python để giới hạn QuerySet ở một số kết quả nhất định. Điều này tương đương với mệnh đề LIMIT và OFFSET của SQL.
- Ví dụ, LIMIT queryset để trả về 5 đối tượng đầu tiên (LIMIT 5):

```
SinhVien.objects.all()[:5]
```

- Ví dụ: LIMIT queryset để trả về các đối tượng thứ 6 đến đối tượng thứ 10 kể từ đối tượng đầu tiên

```
SinhVien.objects.all()[5:10]
```

Lưu ý: Django không hỗ trợ LIMIT queryset theo chỉ mục âm

Tìm kiếm bằng các trường (Fields)

Tìm kiếm bằng trường là cách chúng ta sử dụng phần lớn của mệnh đề WHERE trong SQL. Chúng được chỉ định làm đối số từ khóa **QuerySet filter()**, **exclude()** và **get()**.

Các đối số từ khóa tra cứu cơ bản có dạng **field__lookuptype=value**. (Hai dòng gạch dưới)

```
HoSo.objects.filter(ngay_cap_nhat__lte='2006-01-01')
```

Dịch sang SQL:

```
SELECT * FROM quanli_hoso WHERE ngay_cap_nhat <= '2006-01-01';
```


Trong trường hợp của **ForeignKey** ta có thể sử dụng hậu tố “**_id**” để tra cứu đến đối tượng khóa ngoại này dựa trên **id**

```
SinhVien.objects.filter(lop_dao_tao_id=1)
```

Nếu chúng ta truyền một đối số từ khóa không hợp lệ, hàm tra cứu sẽ tăng `TypeError`.

Tìm kiếm chính xác (exact):

- Tìm kiếm chính xác sẽ tìm kiếm chính xác dựa trên đối số từ khóa đầu vào. Bao gồm cả phân biệt in hoa và in thường.

```
SinhVien.objects.get(ho_va_ten__exact="cau vong")
```

- Trong trường hợp tìm kiếm mà không phân biệt chữ hoa hay thường ta sử dụng **__iexact**:

```
SinhVien.objects.get(ho_va_ten__iexact="cau vong")
```

- Trường hợp cần tìm kiếm với sự có mặt của đối số từ khóa (Áp dụng đối với chữ in hoa và chữ thường):

```
SinhVien.objects.get(ho_va_ten__contains="Tran")
```

Lưu ý:

- **__contains** sẽ truy vấn thành công trong trường hợp “Tran Van A” và không truy vấn thành công trong trường hợp “tran Van A”.
- Nếu muốn truy vấn thành công trong trường hợp còn lại ta sử dụng **__icontains**.

Truy vấn mở rộng thông qua các mối quan hệ:

- Truy vấn thông qua **ForeignKey**:

- Django cung cấp cho chúng ta cách truy vấn mạnh mẽ và trực quan thông qua việc tra cứu các mối quan hệ (Tự động xử lý SQL JOINS).
- Ví dụ: Lấy tất cả các sinh viên thuộc lớp ANTT2017:

```
SinhVien.objects.filter(lop__ten_lop="ANTT2017")
```

- Ngoài ra chúng ta vẫn có thể truy vấn thêm mức độ dựa trên các mối quan hệ khác và đồng thời cũng có thể sử dụng các kỹ thuật truy vấn trước đó.
- Truy vấn thông qua **ManyToManyField** hoặc đảo ngược **ForeignKey**:
 - Mọi dữ kiện bên trong một **filter()** sẽ được áp dụng đồng thời để lọc ra các mục phù hợp với tất cả các dữ kiện đặt ra. Các **filter()** kế tiếp sẽ hạn chế hơn nữa số lượng đối tượng của query, nhưng đối với những quan hệ đa giá trị, chúng sẽ được áp dụng cho bất kỳ đối tượng nào được liên kết với mô hình chính, không nhất thiết là những đối tượng đã được chọn bởi lệnh gọi **filter()** trước đó.

```
SinhVien.objects.filter(lop__ten_lop="ANTT2017").filter(lop__nam=2008)
```

Filter có thể tham chiếu đến các trường của model:

- Nếu chúng ta muốn so sánh giá trị của một trường của model với một trường khác trên cùng một model thì ta sử dụng biểu thức F(). F() hoạt động như một tham chiếu đến trường của model trong một truy vấn. Sau đó, các tham chiếu này có thể được sử dụng trong các filter để so sánh các giá trị của hai trường khác nhau trên cùng một model.
- Ví dụ: Chúng ta cần truy xuất những bài viết có số lượt thích cao hơn số lượt bình luận:

```
BaiViet.objects.filter(luot_thich__gte=F('luot_comment'))
```

Tra cứu phức tạp với Q Objects:

- Thông thường các đối số từ khóa được đưa vào filter sẽ được xem như phép “AND” trong database. Nếu muốn thực hiện truy vấn với phép “OR” ta cần sử dụng **Q objects**.
- Đối tượng **Q** (`django.db.models.Q`) là một đối tượng được sử dụng để đóng gói tập hợp các đối số từ khóa.
- Ví dụ, Q object này đóng gói một **LIKE** query như sau:

```
from django.db.models import Q
Q(ho_va_ten__icontains="Tran")
```

- Các đối tượng **Q** có thể được kết hợp bằng cách sử dụng các toán tử **&** và **|**. Khi một toán tử được sử dụng trên hai đối tượng **Q**, nó sinh ra một đối tượng **Q** mới.
- Ví dụ: câu lệnh này tạo ra một đối tượng **Q** duy nhất đại diện cho "OR" của hai truy vấn "**ho_va_ten__startswith**":

```
Q(ho_va_ten__startswith='Tran') | Q(ho_va_ten__startswith='Nguyen')
```

- Query trên bảng với dòng lệnh sau trong SQL mệnh đề WHERE:

```
WHERE ho_va_ten LIKE 'Tran%' OR ho_va_ten LIKE 'Nguyen%'
```

- Chúng ta có thể tạo các câu lệnh có độ phức tạp tùy ý bằng cách kết hợp các đối tượng **Q** với toán tử **&** và **|** và sử dụng phân nhóm trong ngoặc đơn. Ngoài ra, các đối tượng **Q** có thể được phủ định bằng cách sử dụng toán tử **~**, cho phép tìm kiếm kết hợp kết hợp cả truy vấn bình thường và truy vấn phủ định (**NOT**):

```
Q(ho_va_ten__startswith='Tran') | ~Q(ho_va_ten__endswith='B')
```

- Lưu ý: Các hàm tra cứu có thể kết hợp việc sử dụng đối tượng **Q** và đối số từ khóa. Tất cả các đối số được cung cấp cho một hàm tra cứu đều "**AND**" cùng nhau. Tuy nhiên, nếu có đối tượng **Q**, nó phải đứng trước bất kỳ đối số từ khóa nào. Ví dụ:

```
SinhVien.objects.get(
    Q(ho_va_ten__startswith='Tran') | ~Q(ho_va_ten__endswith='B'),
    gpa__gte=7)
```

- Tuy nhiên ví dụ sau sẽ không hợp lệ do đối tượng Q đứng sau đối số từ khóa:

```
SinhVien.objects.get(
    gpa__gte=7,
    Q(ho_va_ten__startswith='Tran') | ~Q(ho_va_ten__endswith='B'))
```

So sánh các đối tượng objects:

Để so sánh hai model với nhau, ta sử dụng toán tử so sánh Python chuẩn, bằng dấu bằng kép: “==”. Ví dụ:

```
sinh_vien == sinh_vien_1
sinh_vien.ho_va_ten == sinh_vien_1.ho_va_ten
```

Xóa đối tượng:

- Để xóa đối tượng ta sử dụng Phương thức delete(). Phương thức này ngay lập tức xóa đi đối tượng và trả về số lượng đối tượng bị xóa với số lần xóa của mỗi loại đối tượng:

```
sinh_vien.delete()
```

- Chúng ta cũng có thể xóa đối tượng thông qua queryset:

```
SinhVien.objects.filter(ho_va_ten="Tran Van A").delete()
```

Lưu ý: Khi Django xóa một đối tượng, theo mặc định, nó mô phỏng hành vi của ràng buộc **SQL ON DELETE CASCADE** - nói cách khác, bất kỳ đối tượng nào có khóa ngoại trỏ đến đối tượng cần xóa sẽ bị xóa cùng với nó.

Cập nhật nhiều đối tượng cùng một lúc:

- Ví dụ tìm tất cả các sinh viên thuộc khóa k12, cập nhật gpa = 3.5

```
SinhVien.objects.filter(khoa="k12").update(gpa=3.5)
```

- Phương thức **update()** được áp dụng ngay lập tức và trả về số hàng được truy xuất bởi truy vấn (có thể không bằng số hàng được cập nhật nếu một số hàng đã có giá trị mới). Hạn chế duy nhất đối với QuerySet đang được cập nhật là nó chỉ có thể truy cập một bảng cơ sở dữ liệu (bảng chính của model).
- Lưu ý rằng phương thức **update()** được chuyển đổi trực tiếp thành một câu lệnh SQL. Sau đó tiến hành cập nhật trực tiếp số lượng lớn. Đặc biệt phương thức này không chạy bất kỳ phương thức **save()** nào trên các model hoặc phát ra các tín hiệu **pre_save** hoặc **post_save** (hệ quả của việc gọi **save()**). Nếu chúng ta muốn lưu mọi mục trong QuerySet và đảm bảo rằng phương thức **save()** được gọi trên mỗi instance, ta không cần bất kỳ hàm đặc biệt nào để xử lý điều này. Chúng ta chỉ cần tiến hành lặp và gọi phương thức **save()** cho mỗi instance:

```
for s in ds_sv:  
    s.save()
```

Truy vấn trên quan hệ One-to-many:

- **Truy vấn cùng chiều:**
 - Nếu một model có ForeignKey, các instance của model đó sẽ có quyền truy cập vào đối tượng (foreign) liên quan thông qua một thuộc tính của model. Ví dụ:

```
s = SinhVien.objects.filter(id=1)  
s.lop_dao_tao # Trả về đối tượng liên quan của lớp đào tạo
```

- Chúng ta có thể **get** và **set** thông qua các thuộc tính khóa ngoại. Lưu ý, các thay đổi đối với khóa ngoại không được lưu vào cơ sở dữ liệu cho đến khi phương thức **save()** được gọi. Ví dụ:

```
s = SinhVien.objects.filter(id=1)
s.lop_dao_tao = lop_dao_tao_2017
s.save()
```

- Khi truy cập đến đối tượng khóa ngoại Django sẽ chỉ thực sự thực hiện truy vấn trong lần đầu tiên, các lần sau sẽ sử dụng cache data thay vì tạo truy vấn mới. Ví dụ:

```
s = SinhVien.objects.filter(id=1)
print(s.lop_dao_tao) # Thực hiện truy vấn đến database
print(s.lop_dao_tao) # Sử dụng cache
```

- **Truy vấn ở chiều ngược lại**

Nếu model có **ForeignKey**, các instance của model chứa khóa ngoại sẽ có quyền truy cập vào một **Manager** trả về tất cả các instance của model đầu tiên. Theo mặc định, Manager này được đặt tên là **FOO_set**, trong đó **FOO** là tên của model nguồn, viết thường. **Manager** này trả về **QuerySets**, có thể được lọc và thao tác như được mô tả trong phần “Truy vấn” ở trên.

```
lp = LopDaoTao.objects.get(id=1)
lp.sinhvien_set.all() # Trả về tất cả các sinh viên liên quan
```

Truy vấn trên mối quan hệ Many-to-many:

- Cả hai đầu của mối quan hệ many-to-many đều nhận được quyền truy cập API tự động vào đầu còn lại. API hoạt động tương tự như mối quan hệ **one-to-many** “**chiều ngược**” như ở trên.

- Một điểm khác biệt nữa đó là ở cách đặt tên thuộc tính: Tại model xác định **Many-ToManyField** sẽ sử dụng tên thuộc tính của chính trường đó, trong khi model “**đảo ngược**” sử dụng tên model viết thường dựa trên model gốc, cộng với “**__set**” (giống như phần **truy vấn ở chiều ngược ở one-to-many**). Ví dụ:

```
s = SinhVien.objects.get(id=3)
s.lop_hoc.all() # Trả về tất cả lớp học sinh viên này tham gia.
s.lop_hoc.count()
s.lop_hoc.filter(ma_phong='A101')

lp = Lop.objects.get(id=5)
lp.sinhvien_set.all() # Trả về tất cả các sinh viên tham gia lớp học này
```

Truy vấn trên mối quan hệ One-to-one:

- Mối quan hệ **one-to-one** rất giống với mối quan hệ **many-to-one**. Nếu chúng ta xác định **OneToOneField** trên model, các instance của model đó sẽ có quyền truy cập vào đối tượng liên quan thông qua một thuộc tính của model. Ví dụ:

```
class HoSo(models.Model):
    sinhvien = models.OneToOneField(SinhVien, on_delete=models.CASCADE)
    thongtin = models.TextField()

hs = HoSo.objects.get(id=1)
hs.sinhvien # Trả về sinh viên Liên quan
```

- Đối với truy vấn ở chiều ngược lại ta làm tương tự chiều thuận, ví dụ:

```
sinhvien = SinhVien.objects.get(id=1)
sv.hoso # Trả về hồ sơ Liên quan
```

Lưu ý: Nếu không có đối tượng nào được gán cho mối quan hệ này, Django sẽ báo **DoesNotExist**.

2.2.3. Django admin site

Một trong những phần mạnh nhất của Django là giao diện admin. Django admin site đọc metadata từ các mô hình của chúng ta để cung cấp giao diện nhanh chóng, tập trung vào mô hình, nơi người dùng đáng tin cậy có thể quản lý nội dung trên trang web.

2.2.3.1. Model admin

Lớp `ModelAdmin` là đại diện của một model trong giao diện admin. Thông thường, chúng được lưu trữ trong một tệp có tên **admin.py** trong ứng dụng. Xét một ví dụ về **ModelAdmin**:

```
from django.contrib import admin
from .models import SinhVien

# Register your models here.

class SinhVienAdmin(admin.ModelAdmin):
    pass

admin.site.register(SinhVien, SinhVienAdmin)
```

register decorator (`@admin.register(model_name)`):

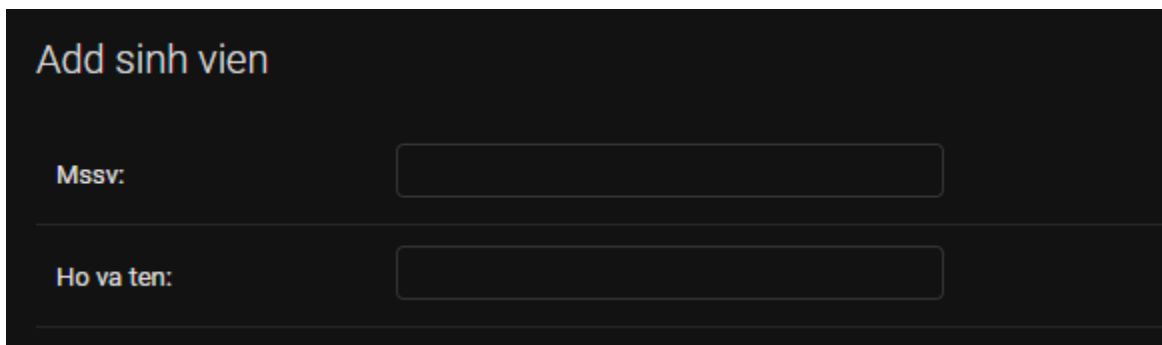
Chúng ta có ví dụ decorator cho ModelAdmin class như sau:


```
from django.contrib import admin
from .models import SinhVien

# Register your models here.

@admin.register(SinhVien)
class SinhVienAdmin(admin.ModelAdmin):
    pass
```

Kết quả của model SinhVien trên admin site:



The screenshot shows the Django admin interface for adding a new student. The title is 'Add sinh vien'. There are two input fields: 'Mssv:' and 'Ho va ten:'. The 'Mssv:' field is on the first line, and the 'Ho va ten:' field is on the second line. Both fields are empty and have a light blue border.

Khi model có hàng chục trường, chúng ta có thể muốn chia biểu mẫu thành các tập trường:

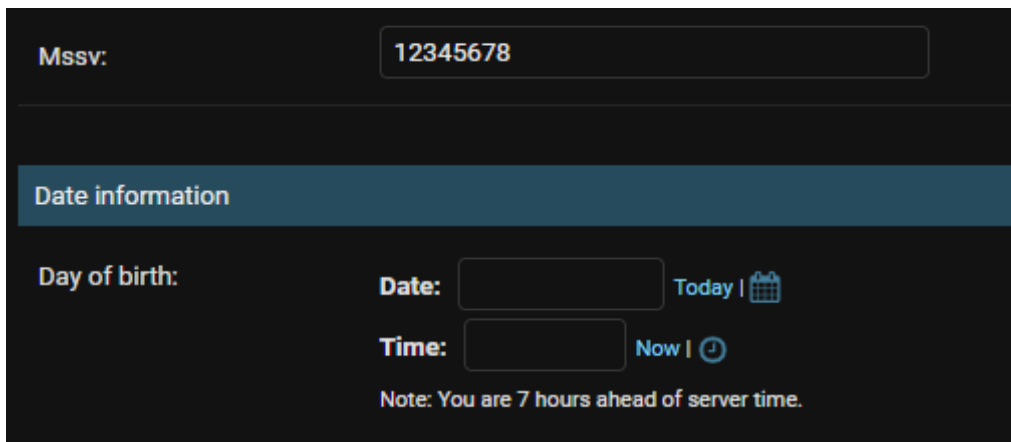
```

from django.contrib import admin
from .models import SinhVien

# Register your models here.
@admin.register(SinhVien)
class SinhVienAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {"fields": ('mssv',)}),
        ('Date information', {'fields': ('day_of_birth',)}),
    )

```

Kết quả thay đổi như sau:



2.3.2. Tùy chỉnh hiển thị nội dung

Theo mặc định, Django hiển thị `str()` của mỗi đối tượng. Nhưng đôi khi sẽ hữu ích hơn nếu chúng ta có thể hiển thị các trường riêng lẻ. Để làm điều đó, hãy sử dụng tùy chọn quản trị `list_display`, là một loạt các tên trường để hiển thị, dưới dạng cột, trên trang danh sách thay đổi cho đối tượng:

```

from django.contrib import admin
from .models import SinhVien

# Register your models here.
@admin.register(SinhVien)
class SinhVienAdmin(admin.ModelAdmin):

    list_display = ('mssv', 'ho_va_ten', 'create_at', 'update_at',
)

```

Kết quả sau thay đổi:

Action:	<input type="text"/>	Go	0 of 1 selected
<input type="checkbox"/>	MSSV	HO VA TEN	
<input type="checkbox"/>	12345678	Tuan Tran Quoc	

CREATE AT	UPDATE AT
Aug. 14, 2021, 4:32 p.m.	Aug. 14, 2021, 4:22 p.m.

Để thực hiện việc lọc các trường ta sử dụng **list_filter**:

```

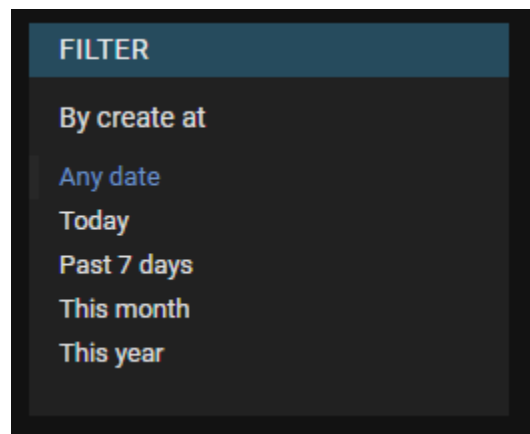
from django.contrib import admin
from .models import SinhVien

# Register your models here.
@admin.register(SinhVien)
class SinhVienAdmin(admin.ModelAdmin):

    list_filter = ('create_at',)

    list_display = ('mssv', 'ho_va_ten', 'create_at', 'update_at',)

```



Để thực hiện việc tìm kiếm ta sử dụng **search_fields**:

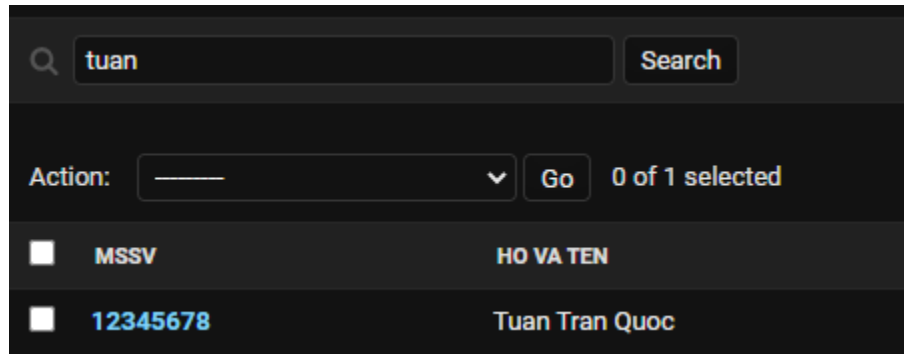
```

from django.contrib import admin
from .models import SinhVien

# Register your models here.
@admin.register(SinhVien)
class SinhVienAdmin(admin.ModelAdmin):

    search_fields = ('ho_va_ten', 'mssv',)

```



The screenshot shows a web application interface. At the top, there is a search bar with the text 'tuan' and a 'Search' button. Below the search bar, there is an 'Action:' label, a dropdown menu, a 'Go' button, and the text '0 of 1 selected'. Below this, there is a table with two rows of data. The first row has a checkbox, the text 'MSSV', and the text 'HO VA TEN'. The second row has a checkbox, the text '12345678', and the text 'Tuan Tran Quoc'.

	MSSV	HO VA TEN
<input type="checkbox"/>	12345678	Tuan Tran Quoc

2.3. Xử lý HTTP requests

2.3.1. URL dispatcher

Khi người dùng yêu cầu một trang từ trang web do Django cung cấp, bên dưới đây là thuật toán mà hệ thống tuân theo để xác định mã Python nào sẽ thực thi

1. Django xác định mô-đun **URLconf** gốc để sử dụng. Thông thường, đây là giá trị của cài đặt **ROOT_URLCONF**, nhưng nếu đối tượng **HttpRequest** đến có thuộc tính **urlconf** (thiết lập bởi middleware), giá trị của nó sẽ được sử dụng thay cho **ROOT_URLCONF** setting.
2. Django load mô-đun Python đó và tìm kiếm các biến `urlpatterns`. Đây phải là một chuỗi của `django.urls.path()` và / hoặc `django.urls.re_path()` instance.
3. Django chạy qua từng **URL pattern**, theo thứ tự và dừng lại ở mẫu đầu tiên phù hợp với requested **URL**, khớp với **path_info**.
4. Khi một trong các **URL pattern** khớp, Django imports và gọi **view** đã có (một hàm Python class-based hoặc function-based view). View được truyền vào các đối số sau:
 - Một instance của `HttpRequest`.
 - Nếu **URL pattern** phù hợp không chứa nhóm được đặt tên, thì các regular expression được cung cấp như là một đối số vị trí.
 - Các đối số từ khóa được tạo thành từ bất kỳ phần được đặt tên phù hợp với biểu thức đường dẫn, được ghi đè bởi bất kỳ đối số nào được xác định bởi

đối số **kwargs** tùy chọn thành **django.urls.path()** hoặc **django.urls.re_path()**.

Nếu không có URL pattern nào phù hợp hoặc nếu bất kì exception nào được bật lên trong thời điểm của quá trình này, Django sẽ gọi một view xử lý lỗi thích hợp.

Ví dụ:

```
from django.urls import path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),
    path('articles/<int:year>', views.year_archive),
    path('articles/<int:year>/<int:month>',
views.month_archive),
    path('arti-
cles/<int:year>/<int:month>/<slug:slug>', views.article_detail),
]
```

Lưu ý:

- Để bắt một giá trị từ URL, hãy sử dụng dấu ngoặc nhọn.
- Các giá trị bắt được có thể tùy chọn bao gồm chuyển đổi loại. Ví dụ: sử dụng **<int:name>** để nắm bắt một tham số số nguyên. Nếu không có trình chuyển đổi, thì bất kỳ chuỗi nào, ngoại trừ ký tự “/”, đều được khớp.
- Không cần thêm dấu gạch chéo ở đầu, bởi vì mọi URL đều có. Ví dụ: **articles**, không phải **/articles**.

Ví dụ:

- Một request tới **/article/2005/03/** sẽ phù hợp với mục thứ ba trong danh sách **URL pattern** nằm trong ví dụ trên. Django tiếp đến sẽ gọi hàm **views.month_archive(request, year = 2005, month = 3)**.
- **/articles/2003/** sẽ phù hợp với mục đầu tiên trong danh sách URL pattern. . Django tiếp đến sẽ gọi hàm **views.special_case_2030**.
- **/articles/2003** sẽ không phù hợp bởi bất kì URL pattern nào.

2.3.1.1. Sử dụng regular expressions

Nếu đường dẫn và cú pháp của trình chuyển đổi không đủ để xác định các URL pattern của chúng ta, ta cũng có thể sử dụng **regular expressions**. Để làm như vậy, ta sử dụng **re_path()** thay vì **path()**.

Trong Python **regular expressions**, cú pháp cho các nhóm **regular expressions** được đặt tên là **(? P <name> pattern)**, trong đó tên là tên của nhóm và mẫu là một số mẫu phù hợp.

Ví dụ:

```

from django.urls import path, re_path

from . import views

urlpatterns = [
    path('articles/2003/', views.special_case_2003),

    re_path(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),

    re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),

    re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<slug>[\w-]+)/$', views.article_detail),
]

```

URLconf tìm kiếm điều gì?

- URLconf tìm kiếm theo URL được yêu cầu, như một chuỗi Python bình thường. Điều này không bao gồm các tham số GET hoặc POST hoặc tên miền.
- Ví dụ:
 - Trong một request tới *https://www.example.com/myapp/*, **URLconf** sẽ tìm **myapp/**.
 - Trong một request tới *https://www.example.com/myapp/?page=3*, **URLconf** sẽ tìm kiếm **myapp/**.
- **URLconf** không xem xét request method. Nói cách khác, tất cả các request method - POST, GET, HEAD, v.v. - sẽ được chuyển đến cùng một hàm cho cùng một **URL**.

2.3.1.2. Chỉ định các giá trị đối số cho view

Ví dụ:

```
# URLconf
from django.urls import path
from . import views
urlpatterns = [
    path('blog/', views.page),
    path('blog/page<int:num>/', views.page),
]

# View (blog/views.py)
def page(request, num=1):
    # Nhận được giá trị của biến num thông qua URL pattern
    ...
```

Trong ví dụ trên, cả hai **URL pattern** đều trỏ đến cùng một view - **views.page** - nhưng mẫu đầu tiên **capture** bất kỳ giá trị gì từ **URL**. Nếu mẫu đầu tiên khớp, hàm `page()` sẽ sử dụng đối số mặc định của nó cho `num = 1`. Nếu mẫu thứ hai khớp, **page()** sẽ sử dụng giá trị `num` được cung cấp bởi URL pattern.

2.3.1.3. Thêm các URLconfs khác

Tại bất kỳ thời điểm nào, **urlpatterns** của chúng ta có thể “**include**” các mô-đun **URLconf** khác. Về cơ bản “root” một tập hợp các **URL** bên dưới các **URL** khác.

Ví dụ:

```

from django.urls import include, path
urlpatterns = [
    # ... snip ...
    path('community/', include('aggregator.urls')),
    path('contact/', include('contact.urls')),
    # ... snip ...
]

```

Khi Django gặp phải **include()**, nó sẽ cắt bất kỳ phần nào của URL khớp với thời điểm đó và gửi chuỗi còn lại đến **URLconf** được bao gồm để xử lý.

Một khả năng khác là bao gồm thêm các URL pattern bổ sung bằng cách sử dụng danh sách các **path() instance**. Ví dụ:

```

from django.urls import include, path
from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    path('reports/', credit_views.report),
    path('reports/<int:id>', credit_views.report),
    path('charge/', credit_views.charge),
]

urlpatterns = [
    path('', main_views.homepage),
    path('help/', include('apps.help.urls')),
    path('credit/', include(extra_patterns)),
]

```

Trong ví dụ trên, `/credit/reports/` URL sẽ được xử lý bởi `credit_views.report()` Django view.

2.3.1.4. Captured parameters từ URL pattern

Một **URLconf** được bao gồm, nhận bất kỳ tham số nào được bắt từ **URLconf** parent, vì vậy ví dụ sau là hợp lệ:

```
# Trong settings/urls/main.py
from django.urls import include, path

urlpatterns = [
    path('<username>/blog/', include('foo.urls.blog')),
]

# Trong foo/urls/blog.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.blog.index),
    path('archive/', views.blog.archive),
]
```

Trong ví dụ trên, biến "username" đã được chuyển đến **URLconf**.

2.3.2. Xây dựng view

Một view đơn giản trả về ngày và giờ hiện tại, dưới dạng tài liệu HTML::

```

from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)

```

Giải thích:

- Đầu tiên, chúng ta import **HttpResponse** class từ mô-đun **django.http**, cùng với thư viện **datetime** của Python.
- Tiếp theo, chúng ta định nghĩa một hàm có tên là **current_datetime**. Đây là hàm view. Mỗi hàm view lấy một đối tượng **HttpRequest** làm tham số đầu tiên của nó, thường được đặt tên là **request**.
- Lưu ý rằng tên của hàm view không quan trọng. Chúng ta đặt tên hàm là **current_datetime** ở đây vì tên đó chỉ rõ chức năng của hàm.
- View trả về một đối tượng **HttpResponse** có chứa phản hồi đã tạo. Mỗi view function có trách nhiệm trả về một đối tượng **HttpResponse**. (Có những trường hợp ngoại lệ)

2.3.2.1. Trả về lỗi

Django cung cấp trợ giúp để trả lại mã lỗi **HTTP**. Có các lớp con của **HttpResponse** cho một số mã trạng thái **HTTP** phổ biến khác 200 (nghĩa là “OK”). Chúng ta có thể tìm thấy danh sách đầy đủ các lớp con có sẵn trong tài liệu request/response. Trả về một instance của một trong những lớp con đó thay vì một **HttpResponse** bình thường để thông báo lỗi. Ví dụ:

```

from django.http import HttpResponse, HttpResponseNotFound

def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')

```

Ngoài ra chúng ta còn có thể trả về status code:

```

from django.http import HttpResponse

def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)

```

2.3.2.2. *Http404 exception*

Khi chúng ta trả về một lỗi, chẳng hạn như **HttpResponseNotFound**, ta có thể xác định trang HTML cho kết quả lỗi trả về:

```

return HttpResponseNotFound('<h1>Page not found</h1>')

```

Để thuận tiện chúng ta nên có một trang lỗi 404 nhất quán trên trang web của mình, Django cung cấp một trang thông báo lỗi Http404. Nếu raise Http404 tại bất kỳ điểm nào trong một hàm view, Django sẽ trả về một trang lỗi chuẩn cho ứng dụng, cùng với mã lỗi HTTP 404.

```

from django.http import Http404
from django.shortcuts import render
from polls.models import Poll

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render(request, 'polls/detail.html', {'poll': p})

```

2.3.3. View decorators

Cho phép sử dụng các phương thức HTTP:

Các decorators trong **django.views.decorators.http** có thể được sử dụng để hạn chế việc truy cập vào các view dựa trên request method. Các decorators này sẽ trả về **django.http.HttpResponseNotAllowed** nếu các điều kiện không được đáp ứng.

Decorator yêu cầu một view chỉ chấp nhận các request method cụ thể:

```

from django.views.decorators.http import require_http_methods

@require_http_methods(["GET", "POST"])
def my_view(request):
    # Chỉ có GET hoặc POST request được chấp nhận
    # ...
    pass

```

- **request_GET()**: Decorator để yêu cầu view chỉ chấp nhận phương thức GET.
- **request_POST()**: Decorator để yêu cầu view chỉ chấp nhận phương thức POST.

- **request_safe()**: Decorator để yêu cầu view chỉ chấp nhận các phương thức GET và HEAD. Các phương thức trên được cho là an toàn vì không làm thay đổi trạng thái hệ thống.

2.3.4. File Uploads

2.3.4.1. File uploads cơ bản

Xét một form có chứa `FileField`:

```
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

Một view xử lý form này sẽ nhận dữ liệu tệp trong **request.FILES**, là một dictionary chứa khóa cho mỗi **FileField** (hoặc **ImageField**, hoặc lớp con **FileField** khác) trong form. Vì vậy, dữ liệu từ form trên sẽ có thể truy cập được dưới dạng **request.FILES['file']**.

Lưu ý rằng **request.FILES** sẽ chỉ chứa dữ liệu nếu phương thức yêu cầu là **POST**, ít nhất một trường file thực sự đã được post và **<form>** được gửi phải có thuộc tính **enctype="multipart/form-data"**. Nếu không, **request.FILES** sẽ trống.

Ví dụ:

```

from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm

# Giả sử chúng ta có hàm xử lý sau khi upload thành công
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
        else:
            form = UploadFileForm()
    return render(request, 'upload.html', {'form': form})

```

2.3.4.1.1. Xử lý upload với model

Nếu chúng ta đang lưu tệp trên Model bằng **FileField**, thì việc sử dụng **ModelForm** làm cho quá trình này dễ dàng hơn nhiều. Đối tượng tệp sẽ được lưu vào vị trí được chỉ định bởi đối số **upload_to** của **FileField** tương ứng khi gọi **form.save()**:


```

from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import ModelFormWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = ModelFormWithFileField(request.POST, request.FILES
)

        if form.is_valid():
            # file đã được lưu
            form.save()
            return HttpResponseRedirect('/success/url/')
        else:
            form = ModelFormWithFileField()
            return render(request, 'upload.html', {'form': form})

```

2.3.4.1.2. Upload nhiều file

Nếu chúng ta muốn tải lên nhiều tệp bằng một form field, đặt thuộc tính **multiple** của HTML của form widget:

```

from django import forms

class FileFieldForm(forms.Form):
    file_field = forms.FileField(
        widget=forms.ClearableFileInput(attrs={'multiple': True})
    )

```

Sau đó, ghi đè phương thức POST của lớp con FormView để xử lý nhiều tệp tải lên:

```

from django.views.generic.edit import FormView
from .forms import FileFieldForm

class FileFieldFormView(FormView):
    form_class = FileFieldForm
    template_name = 'upload.html' # Thay bằng template đang dùng
    .
    success_url = '...' # Thay bằng URL hợp lệ khi POST thành công.

    def post(self, request, *args, **kwargs):
        form_class = self.get_form_class()
        form = self.get_form(form_class)
        files = request.FILES.getlist('file_field')
        if form.is_valid():
            for f in files:
                ... # Ở đây chúng ta có từng file loop trong
                    # danh sách các file POST
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

```

2.3.4.2. Xử lý upload

request.POST được truy cập bởi **CsrfViewMiddleware** được theo mặc định. Điều này có nghĩa là chúng ta sẽ cần sử dụng **csrf_exempt()** trên view của mình để cho phép thay đổi trình xử lý file upload. Sau đó, ta sẽ cần sử dụng **csrf_protect()** trên hàm thực sự xử lý request. Lưu ý rằng điều này có nghĩa là trình xử lý có thể bắt đầu nhận tệp tải lên trước khi quá trình kiểm tra **CSRF** được thực hiện. Ví dụ:

```

from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
    return _upload_file_view(request)

@csrf_protect
def _upload_file_view(request):
    ... # Tiến hành xử lý request

```

Xử lý lưu trữ:

Django đi kèm với lớp **django.core.files.storage.FileSystemStorage** triển khai lưu trữ tệp hệ thống tệp cục bộ cơ bản.

Ví dụ: mã sau sẽ lưu trữ các tệp được tải lên trong **/media/photos** bất kể cài đặt của **MEDIA_ROOT** là gì:

```

from django.core.files.storage import FileSystemStorage
from django.db import models

fs = FileSystemStorage(location='/media/photos')

class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)

```

2.3.4.3. Các hàm rút gọn

2.3.4.3.1. Hàm `render()`

Kết hợp template nhất định với một context dictionary nhất định và trả về một đối tượng **HttpResponse** với text được hiển thị.

Cách sử dụng:

render(request, template_name, context=None, content_type=None, status=None, using=None)

Đối số tùy chọn:

- **context**: Từ điển các giá trị để thêm vào ngữ cảnh mẫu. Theo mặc định, đây là một từ điển trống. Nếu một giá trị trong từ điển có thể gọi được, thì dạng xem sẽ gọi nó ngay trước khi hiển thị mẫu.
- **content_type**: Loại MIME để sử dụng cho tài liệu kết quả. Mặc định là 'text / html'.
- **status**: Mã trạng thái cho phản hồi. Mặc định là 200.
- **using**: NAME của một mẫu template sử dụng để load template.

Ví dụ:

```
from django.shortcuts import render

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html', {
        'foo': 'bar',
    }, content_type='application/xhtml+xml')
```

2.3.4.3.2. Hàm `redirect()`

Trả về một **HttpResponseRedirect** đến URL thích hợp cho các đối số được truyền vào.

Các đối số có thể là:

- Model: hàm `get_absolute_url ()` của mô hình sẽ được gọi.
- view name: `reverse ()` sẽ được sử dụng để phân giải ngược tên.
- URL tuyệt đối hoặc tương đối, sẽ được sử dụng cho vị trí chuyển hướng.
- Theo mặc định đưa ra một chuyển hướng tạm thời; truyền vào `permanent=True` để đưa ra chuyển hướng vĩnh viễn.

Ví dụ:

```
def my_view(request):
    ...
    return redirect('/some/url/')
```

```
def my_view(request):
    ...
    return redirect('https://example.com/')
```

2.3.5. Generic views

2.3.5.1. Base views

Ba lớp sau cung cấp nhiều chức năng cần thiết để tạo Django views. Chúng ta có thể coi chúng là dạng xem parent, có thể được sử dụng bởi chính chúng hoặc được thừa kế từ đó.

2.3.5.1.1. View

Tất cả các class-based view đều được kế thừa từ base class. Nó không hoàn toàn là một generic view và do đó cũng có thể được import từ **django.views**.

Phương thức:

- **setup():**
 - Thực hiện khởi tạo chế độ xem khóa trước khi *dispatch()*.
 - Nếu ghi đè phương thức này, ta phải gọi lệnh *super()*.
- **dispatch():**

- Phần view của view - phương thức chấp nhận một đối số yêu cầu cộng với các đối số và trả về một phản hồi **HTTP**.
- Việc triển khai mặc định sẽ kiểm tra phương thức **HTTP** và cố gắng ủy quyền cho một phương thức phù hợp với phương thức **HTTP**; một **GET** sẽ được ủy quyền để **get()**, một **POST** để **post()**, v.v.
- Theo mặc định, một **HEAD** request sẽ được ủy quyền để **get()**. Nếu chúng ta cần xử lý các **HEAD** request theo cách khác với **GET**, ta có thể ghi đè phương thức **head()**.
- **http_method_not_allowed():**
 - Nếu view được gọi bằng một phương thức HTTP không được hỗ trợ, thì phương thức **HttpResponseNotAllowed** sẽ được gọi thay thế.
- **options():**
 - Xử lý phản hồi các request cho **OPTIONS HTTP verb**. Kết quả trả về phản hồi với tiêu đề Cho phép chứa danh sách tên phương thức **HTTP** cho phép của view.

Ví dụ:

views.py

```
from django.http import HttpResponseRedirect
from django.views import View

class MyView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponseRedirect('Hello, World!')
```

urls.py

```
from django.urls import path

from myapp.views import MyView

urlpatterns = [
    path('mine/', MyView.as_view(), name='my-view'),
]
```

2.3.5.1.2. *TemplateView*

Hiển thị template, với context chứa trong các parameters bắt được từ URL.

Các phương thức được sử dụng:

- **setup()**
- **dispatch()**
- **http_method_not_allowed()**
- **get_context_data()**

Ví dụ:

views.py

```

from django.views.generic.base import TemplateView

from articles.models import Article

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['latest_articles'] = Article.objects.all()[:5]
        return context

```

urls.py

```

from django.urls import path

from myapp.views import HomePageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]

```

2.3.5.1.3. RedirectView

Chuyển hướng đến một URL nhất định.

URL đã cho có thể chứa định dạng chuỗi kiểu từ điển, định dạng này sẽ được nội suy dựa trên các tham số bắt được trong URL. Bởi vì nội suy từ khóa luôn được thực hiện (ngay cả khi không có đối số nào được chuyển vào), bất kỳ ký tự "%" nào trong URL phải được viết là "%" để Python sẽ chuyển đổi chúng thành một dấu phân trăm duy nhất trên đầu ra.

Nếu URL đã cho là **None**, Django sẽ trả về **HttpResponseGone** (410).

Các phương thức được sử dụng:

- **setup()**
- **dispatch()**
- **http_method_not_allowed()**
- **get_context_data()**

Ví dụ:

views.py

```
from django.shortcuts import get_object_or_404
from django.views.generic.base import RedirectView

from articles.models import Article

class ArticleCounterRedirectView(RedirectView):

    permanent = False
    query_string = True
    pattern_name = 'article-detail'

    def get_redirect_url(self, *args, **kwargs):
        article = get_object_or_404(Article, pk=kwargs['pk'])
        article.update_counter()
        return super().get_redirect_url(*args, **kwargs)
```

urls.py

```

from django.urls import path
from django.views.generic.base import RedirectView

from article.views import ArticleCounterRedirectView, ArticleDetail
    ilView

urlpatterns = [
    path('counter/<int:pk>/', ArticleCounterRedirectView.as_view(
), name='article-counter'),
    path('details/<int:pk>/', ArticleDetailView.as_view(), name='
article-detail'),
    path('go-to-
django/', RedirectView.as_view(url='https://www.djangoproject.com
/'), name='go-to-django'),
]

```

2.3.5.2. *Generic display views*

2.3.5.2.1. *DetailView*

- Các phương thức sử dụng:
- `setup()`
- `dispatch()`
- `http_method_not_allowed()`
- `get_template_names()`
- `get_slug_field()`
- `get_queryset()`
- `get_object()`
- `get_context_object_name()`
- `get_context_data()`

- `get()`
- `render_to_response()`

Ví dụ:

myapp/views.py:

```
from django.utils import timezone
from django.views.generic.detail import DetailView

from articles.models import Article

class ArticleDetailView(DetailView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

myapp/urls.py:

```
from django.urls import path

from article.views import ArticleDetailView

urlpatterns = [
    path('<slug:slug>/', ArticleDetailView.as_view(), name='article-detail'),
]
```

myapp/article_detail.html:

```
<h1>{{ object.headline }}</h1>
<p>{{ object.content }}</p>
<p>Reporter: {{ object.reporter }}</p>
<p>Published: {{ object.pub_date|date }}</p>
<p>Date: {{ now|date }}</p>
```

2.3.5.2.2. *ListView*

Các phương thức được sử dụng:

- **setup()**
- **dispatch()**
- **http_method_not_allowed()**
- **get_template_names()**
- **get_queryset()**
- **get_context_object_name()**
- **get_context_data()**
- **get()**
- **render_to_response()**

Ví dụ:

views.py:

```
from django.utils import timezone
from django.views.generic.list import ListView

from articles.models import Article

class ArticleListView(ListView):

    model = Article
    paginate_by = 100 # Nếu muốn phân trang

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

myapp/urls.py:

```
from django.urls import path

from article.views import ArticleListView

urlpatterns = [
    path('', ArticleListView.as_view(), name='article-list'),
]
```

myapp/article_detail.html:

```

<h1>Articles</h1>
<ul>
{% for article in object_list %}
    <li>{{ article.pub_date|date }} - {{ article.headline }}</li>
{% empty %}
    <li>No articles yet.</li>
{% endfor %}
</ul>

```

2.3.5.3. Generic editing views

2.3.5.3.1. FormView

Một view hiển thị một form. Khi bị lỗi, hiển thị lại form có lỗi xác thực; nếu thành công, chuyển hướng đến một URL mới.

Ví dụ:

myapp/forms.py:

```

from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # Gửi email sử dụng self.cleaned_data dictionary
        pass

```

myapp/views.py:

```

from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactFormView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # Phương thức này được gọi khi form POST hợp lệ.
        # Nên trả về HttpResponseRedirect.
        form.send_email()
        return super().form_valid(form)

```

myapp/contact.html:

```

<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Send message">
</form>

```

2.3.5.3.2. CreateView

Một view hiển thị form để tạo đối tượng, hiển thị lại biểu mẫu có lỗi xác thực (nếu có) và lưu đối tượng.

Ví dụ:

myapp/views.py:

```
from django.views.generic.edit import CreateView
from myapp.models import Author
```

```
class AuthorCreateView(CreateView):
    model = Author
    fields = ['name']
```

myapp/author_form.html:

```
<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save">
</form>
```

2.3.5.3.3. UpdateView

Một view hiển thị form để chỉnh sửa đối tượng hiện có, hiển thị lại form có lỗi xác thực (nếu có) và lưu các thay đổi đối với đối tượng. Điều này sử dụng form được tạo tự động từ lớp model của đối tượng (trừ khi một lớp biểu mẫu được chỉ định theo cách thủ công).

Ví dụ:

myapp/views.py:

```
from django.views.generic.edit import UpdateView
from myapp.models import Author
```

```
class AuthorUpdateView(UpdateView):
    model = Author
    fields = ['name']
    template_name_suffix = '_update_form'
```

myapp/author_update_form.html:


```
<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Update">
</form>
```

2.3.5.3.4. DeleteView

Một view hiển thị trang xác nhận và xóa một đối tượng hiện có. Đối tượng đã cho sẽ chỉ bị xóa nếu phương thức yêu cầu là POST. Nếu chế độ xem này được tìm nạp qua GET, nó sẽ hiển thị trang xác nhận phải chứa biểu mẫu POST lên cùng một URL.

Ví dụ:

myapp/views.py:

```
from django.urls import reverse_lazy
from django.views.generic.edit import DeleteView
from myapp.models import Author

class AuthorDeleteView(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

myapp/author_confirm_delete.html:

```
<form method="post">{% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
    <input type="submit" value="Confirm">
</form>
```

2.4. Forms

2.4.1. HTML form

Trong HTML, form là một tập hợp các phần tử bên trong `<form> ... </form>` cho phép người truy cập thực hiện những việc như nhập văn bản, chọn tùy chọn, thao tác các đối tượng hoặc điều khiển, v.v. và sau đó gửi lại thông tin đó đến máy chủ.

Các phần tử trên giao diện của form có thể là văn bản, checkbox, ảnh v.v và được tích hợp vào chính HTML.

Một form bắt buộc phải xác định hai tiêu chí sau:

- **where:** Xác định URL mà dữ liệu tương ứng của user input sẽ được trả về
- **how:** Dữ liệu được trả về bởi phương thức HTTP nào

Ví dụ: Login form của Django Admin chứa các phần tử `<input>`: loại **type** = “**text**” dùng cho username, loại **type** = “**password**” dùng cho password và loại **type** = “**submit**” dùng cho “**Login**” Button. Đồng thời nó cũng chứa các hidden text fields mà người dùng không nhìn thấy, mà Django dùng để xác định việc cần phải làm tiếp theo.

Django login form cũng cho trình duyệt biết rằng dữ liệu của **form** phải được gửi đến **URL** được chỉ định trong thuộc tính action của `<form>` - / **admin** / - và nó sẽ được gửi bằng cơ chế **HTTP** được chỉ định bởi thuộc tính method = post.

Khi phần tử `<input type = "submit" value = "Log in">` được kích hoạt, dữ liệu được trả về / admin /.

GET và POST

GET và **POST** là các phương thức HTTP duy nhất để sử dụng khi xử lý các form.

Django login form được trả về bằng cách sử dụng phương thức **POST**, trong đó trình duyệt gói dữ liệu của form, mã hóa nó cho quá trình truyền, gửi đến máy chủ và sau đó đợi nhận lại phản hồi.

Ngược lại, GET gói dữ liệu đã gửi thành một chuỗi và sử dụng chuỗi này để tạo URL. URL chứa địa chỉ nơi dữ liệu phải được gửi, cũng như các khóa và giá trị dữ liệu. Chúng ta có thể dễ dàng thấy được điều này khi thực hiện tìm kiếm trong Django documentation, URL có dạng:

<https://docs.djangoproject.com/search/?q=forms&release=1>.

GET và **POST** thường được sử dụng cho các mục đích khác nhau.

Bất kỳ request nào được sử dụng để thay đổi trạng thái của hệ thống - ví dụ: một request thực hiện thay đổi trong cơ sở dữ liệu - nên sử dụng **POST**. **GET** chỉ nên được sử dụng cho các yêu cầu không ảnh hưởng đến trạng thái của hệ thống.

GET cũng sẽ không phù hợp với form chứa mật khẩu, vì mật khẩu sẽ xuất hiện trong **URL**, cả trong lịch sử trình duyệt và nhật ký máy chủ, tất cả đều ở dạng văn bản thuần túy. Nó cũng không phù hợp với số lượng lớn dữ liệu hoặc dữ liệu nhị phân, chẳng hạn như hình ảnh. Ứng dụng Web sử dụng **GET** request cho admin form chứa rủi ro bảo mật: Kẻ tấn công có thể dễ dàng bắt chước request của form để giành quyền truy cập vào các phần nhạy cảm của hệ thống. **POST**, cùng với các biện pháp bảo vệ khác như **CSRF** của Django cung cấp nhiều kiểm soát hơn đối với việc truy cập.

2.4.2 Django Form

Django xử lý ba phần riêng biệt của công việc liên quan đến form:

- Chuẩn bị và cấu trúc lại dữ liệu để sẵn sàng hiển thị.
- Tạo các HTML form cho dữ liệu.
- Tiếp nhận và xử lý các form và dữ liệu đã gửi từ client.

Trung tâm của hệ thống các thành phần này chính là lớp Django's Form. Tương tự như Django model mô tả cấu trúc logic của một đối tượng, hành vi của nó và cách các bộ phận của nó được hiển thị cho chúng ta, một Form class mô tả một form và xác định cách nó hoạt động và cách nó hiển thị.

Theo cách tương tự mà các trường của Model class ánh xạ tới các trường cơ sở dữ liệu, các trường của Form class ánh xạ tới các phần tử <input> của form HTML. (Một ModelForm ánh xạ các trường của Model class với các phần tử <input> của HTML form thông qua Form)

Bản thân các trường của form là các lớp; Chúng quản lý dữ liệu của Form và thực hiện xác thực khi form được gửi. DateField và FileField xử lý các loại dữ liệu rất khác nhau và phải xử lý khác nhau với các kiểu dữ liệu này.

Khởi tạo, xử lý và kiết xuất form:

Khi kiết xuất một đối tượng trong Django, chúng ta thường:

- Giữ đối tượng trong một view (chẳng hạn lấy đối tượng từ cơ sở dữ liệu).
- Chuyển đối tượng vào template context.
- Sau đó chuyển sang HTML markup bằng cách sử dụng các form.

Khi chúng ta khởi tạo một Form, ta có thể chọn để trống hoặc điền trước các form, ví dụ với:

- Dữ liệu được lưu từ một Model instance.
- Dữ liệu lấy từ các nguồn khác.
- Dữ liệu nhận được từ form trước đó.

Xây dựng Form:

Xây dựng form trong HTML:

Giả sử chúng ta muốn tạo một form đơn giản trên trang web của mình để lấy tên của người dùng. Ta có thể làm như sau:

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{
current_name }}">
    <input type="submit" value="OK">
</form>
```

Form này yêu cầu trình duyệt gửi form data đến **URL / your-name/**, sử dụng phương thức **POST**. Form này hiển thị một trường text, có nhãn “Your name:” và một submit button. Trong ngữ cảnh Form có chứa biến **current_name**, biến đó sẽ được sử dụng để điền trước trường **your_name**.

Bây giờ, chúng ta sẽ cần một view tương ứng với URL / your-name/ để này sẽ tìm kiếm cặp key/value thích hợp trong request và sau đó xử lý chúng.

Xây dựng form trong Django:

Form class:

```
class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100
)
```

- Form trên được định nghĩa với một trường duy nhất (your_name) và nhãn này sẽ xuất hiện trong **<label>** khi nó được hiển thị.
- Độ dài tối đa cho phép của trường được xác định bởi **max_length**. Khi đặt **maxlength = “max_length”** trên **HTML <input>** (Trình duyệt sẽ ngăn người dùng nhập nhiều hơn số ký tự này). Điều đó cũng có nghĩa là khi Django nhận form submit từ trình duyệt, nó sẽ xác thực độ dài của dữ liệu.

Form instance có một phương thức gọi là **is_valid()**, phương thức này xác thực cho tất cả các trường của form. Khi phương thức được gọi với xác thực data:

- return True
- Đặt dữ liệu của Form trong thuộc tính `clean_data`.

Lưu ý: Form sau khi kết xuất không bao gồm `<form>` tags, hoặc submit button.

Xây dựng view:

Để xử lý Form, chúng ta cần khởi tạo Form trong view cho URL mà chúng tôi muốn trả về:

```
def get_name(request):
    # Kiểm tra POST request
    if request.method == "POST":
        # Tạo một form instance và điền dữ liệu vào request
        form = NameForm(request.POST)
        # Kiểm tra xem form có valid hay không
        if form.is_valid():
            # Xử lý data trong form.cleaned_data
            # Điều hướng đến một URL mới
            return HttpResponseRedirect('/thanks/')
    # Nếu GET trả về form rỗng
    else:
        form = NameForm()
    return render(request, 'name.html', {'form': form})
```

- Khi chúng ta truy cập vào URL với GET request, nó sẽ tạo một Form instance trống và đặt nó trong template context để hiển thị. Đây là những gì chúng ta có thể mong đợi khi lần đầu tiên ta truy cập URL.
- Nếu Form được gửi bằng cách sử dụng POST request, thì view sẽ một lần nữa tạo một Form instance và điền vào nó với dữ liệu từ request: **form = NameForm(request.POST)**, đây được gọi là “liên kết dữ liệu với Form”.

- Tiếp đến chúng ta gọi phương thức **is_valid()** của Form. Nếu không phải **True**, quay lại template với form cũ. Lần này Form không còn trống (không bị ràng buộc) nên HTML Form sẽ được điền với dữ liệu đã được gửi từ trước đó, nơi nó có thể được chỉnh sửa và sửa chữa theo yêu cầu.
- Nếu **is_valid()** là **True**, giờ chúng ta có thể xử dụng dữ liệu đã xác thực trong thuộc tính `clean_data` của Form. Chúng ta có thể sử dụng dữ liệu này để cập nhật cơ sở dữ liệu hoặc thực hiện các xử lý khác trước khi gửi chuyển hướng.

Template:

Tại `name.html`

```
<form method="post" action="/your-name/">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```

Tất cả các trường của Form và thuộc tính của chúng sẽ được giải nén thành HTML markup từ `{{ form }}` bằng Django's template language.

Lưu ý:

- **Forms và Cross Site Request Forgery protection:** Django cung cấp một biện pháp bảo vệ để sử dụng để chống lại các **Cross Site Request Forgery protection** trên trang web. Khi gửi Form thông qua POST method với tính năng bảo vệ CSRF được bật, ta phải sử dụng **csrf_token** như trong ví dụ trên.
- **HTML5 input types and browser validation:** Nếu Form của ta chứa **URLField**, **EmailField** hoặc bất kỳ loại trường **Integer**, Django sẽ sử dụng url, email và number HTML5 input types. Theo mặc định, các trình duyệt có thể áp dụng xác thực của riêng trên các trường này, có thể nghiêm ngặt hơn xác thực của Django. Nếu

chúng muốn vô hiệu hóa nó, đặt thuộc tính **novalidate** trên **form tags** hoặc chỉ định một widget field con khác, như **TextInput**.

Bây giờ chúng ta đang làm việc với web form, được mô tả bởi Django Form, được xử lý bởi view và được hiển thị dưới dạng HTML <form>.

Lặp trong form field:

Nếu chúng ta muốn sử dụng cùng HTML cho mỗi Form Fields, ta có thể giảm số lượng code trùng lặp bằng cách lặp lần lượt qua **Form Field** bằng cách sử dụng vòng lặp **{% for %}**:

```
{% for field in form %}
    {{ field }}
{% endfor %}
```

Các thuộc tính hữu ích trên {{ field }} bao gồm:

Bảng 2. 1 Các thuộc tính trên field

Thuộc tính	Công dụng
label	Nhãn của trường, ví dụ: Địa chỉ email.
label_tag	Nhãn của trường được bọc trong thẻ HTML <label>
id_for_label	ID sẽ được sử dụng cho trường
value	Giá trị của trường. ví dụ: django@example.com.
html_name	Tên của trường sử dụng cho input
help_text	Bất kỳ văn bản trợ giúp nào đã được liên kết với trường này.
errors	Lấy thông báo lỗi nếu có
is_hidden	True nếu trường ẩn, ngược lại False

2.4.3. Formsets

class BaseFormSet

Formset là một lớp trừu tượng để làm việc với nhiều Form trên cùng một trang. Giả sử chúng có form sau:

```
class BaiVietForm(forms.Form):
    tieu_de = forms.CharField()
    ngay_xuat_ban = forms.DateField()
```

Chúng ta có thể muốn cho phép người dùng tạo nhiều bài viết cùng một lúc. Để tạo một formset từ BaiVietForm, ta cần:

```
BaiVietFormSet = formset_factory(BaiVietForm)
```

Bây giờ chúng ta đã tạo một **formset** có tên **BaiVietFormSet**. Khởi tạo **formset** cấp cho ta khả năng lặp lại các form trong formset và hiển thị chúng như cách mà ta làm với form thông thường:

```
formset = BaiVietFormSet()
for form in formset:
    print(form.as_table())
```

Ta có thể thấy, nó chỉ hiển thị một form trống. Số lượng form trống hiển thị được kiểm soát bởi **extra parameter (Tham số phụ)**. Theo mặc định, Formset_factory() định nghĩa một form bổ sung; ví dụ sau sẽ tạo một formset class để hiển thị hai form trống:

```
BaiVietFormSet = formset_factory(BaiVietForm, extra=2)
```

Sử dụng dữ liệu ban đầu với formset:

Dữ liệu ban đầu là những gì thúc đẩy khả năng sử dụng chính của một formset. Như được đề cập ở trên, chúng ta có thể xác định số lượng các form phụ. Điều này có nghĩa là ta đang cho formset biết có bao nhiêu form bổ sung để hiển thị ngoài số form mà nó tạo ra từ dữ liệu ban đầu. xét ví dụ:

```
BaiVietFormSet = formset_factory(BaiVietForm, extra=2, max_num=1)
formset = BaiVietFormSet(initial=[
    {'tieu_de': 'Django',
     'ngay_xuat_ban': datetime.date.today()}])
for form in formset:
    print(form.as_table())
```

Hiện tại có tổng cộng ba form hiển thị ở trên. Một cho dữ liệu ban đầu đã được chuyển vào và hai là các form bổ sung. Cũng lưu ý rằng chúng ta đang đưa vào một danh sách các dictionary làm dữ liệu ban đầu.

Giới hạn số lượng form tối đa:

Tham số **max_num** đối với **formset_factory()** cung cấp cho chúng ta khả năng giới hạn số lượng form mà formset sẽ hiển thị:

```
BaiVietFormSet = formset_factory(BaiVietForm, extra=2, max_num=1)
formset = BaiVietFormSet()
for form in formset:
    print(form.as_table())
```

Nếu giá trị của **max_num** lớn hơn số lượng phần tử hiện có trong dữ liệu ban đầu, thì các form trống bổ sung sẽ được thêm vào formset, miễn là tổng số form không vượt quá **max_num**. Ví dụ: nếu **extra = 2** và **max_num = 2** và formset được khởi tạo với một phần tử ban đầu, một form cho phần tử ban đầu và một form trống sẽ được hiển thị.

Nếu số lượng phần tử trong dữ liệu ban đầu vượt quá **max_num**, tất cả các **form dữ liệu ban đầu** sẽ được hiển thị bất kể giá trị của **max_num** và không có form bổ sung nào được hiển thị. Ví dụ: nếu **extra = 3** và **max_num = 1** và formset được khởi tạo với hai mục ban đầu, thì hai form có dữ liệu ban đầu sẽ được hiển thị.

Giá trị `max_num` không thiết lập (bằng `None`, mặc định) sẽ đặt giới hạn lớn về số lượng form được hiển thị (1000). Trên thực tế, tương đương với không có giới hạn.

Giới hạn số lượng form được khởi tạo tối đa:

Tham số `absolute_max` của `formset_factory()` cho phép giới hạn số lượng form có thể được khởi tạo khi cung cấp dữ liệu POST. Điều này bảo vệ chống lại các **cuộc tấn công cạn kiệt bộ nhớ** bằng cách sử dụng các POST request giả mạo:

```
BaiVietFormSet = formset_factory(BaiVietForm, absolute_max=1500)
data = {
    'form-TOTAL_FORMS': '1501',
    'form-INITIAL_FORMS': '0',
}
formset = BaiVietFormSet(data)
len(formset.forms)
# 1500
formset.is_valid()
# False
formset.non_form_errors()
# ['Please submit at most 1000 forms.']
```

Chứng thực Formset:

- Xác thực với `Formset` gần giống với xác thực một form thông thường. Phương thức **`is_valid`** trên `formset` để cung cấp một cách thuận tiện việc xác thực tất cả các form trong một `Formset`:

```

BaiVietFormSet = formset_factory(BaiVietForm)
data = {
    'form-TOTAL_FORMS': '1',
    'form-INITIAL_FORMS': '0',
}
formset = BaiVietFormSet(data)
formset.is_valid()
# True

```

- Chúng ta truyền vào Formset data rỗng dẫn đến một form hợp lệ. Formset đủ thông minh để bỏ qua các form bổ sung không được thay đổi. Nếu chúng ta truyền vào một bài báo không hợp lệ:

```

BaiVietFormSet = formset_factory(BaiVietForm)
data = {
    'form-0-tieu_de': 'Test',
    'form-0-ngay_xuat_ban': '1904-06-16',
    'form-0-tieu_de': 'Test',
    'form-0-ngay_xuat_ban': '1904-06-16',
    'form-1-tieu_de': 'Test 2',
    'form-1-
ngay_xuat_ban': '', # thời gian bắt buộc không được để trống
}
formset = BaiVietFormSet(data)
formset.is_valid()
# False
formset.errors
# [{}, {'ngay_xuat_ban': ['This field is required.']}]

```

- Như chúng ta có thể thấy, **formset.errors** là một danh sách có các mục tương ứng với các form trong Formset. Việc xác thực đã được thực hiện cho mỗi form trong hai form và thông báo lỗi dự kiến sẽ xuất hiện cho phần tử thứ hai.
- Giống như khi sử dụng **Form** thông thường, mỗi trường trong các Form của Formset có thể bao gồm các thuộc tính HTML như **maxlength** để xác thực trên trình duyệt. Tuy nhiên, các trường form của Formset sẽ không bao gồm thuộc tính bắt buộc vì quá trình xác thực có thể không chính xác trong quá trình thêm và xóa **Form**.

BaseFormSet.total_error_count()

- Để kiểm tra xem có bao nhiêu lỗi trong **Formset**, chúng ta có thể sử dụng phương thức **total_error_count**:

```
formset.errors
# [{}, {'ngay_xuat_ban': ['This field is required.']}]
len(formset.errors)
# 2
formset.total_error_count()
# 1
```

- Chúng ta cũng có thể kiểm tra xem dữ liệu của form có khác với dữ liệu ban đầu hay không (tức là form đã được gửi mà không có bất kỳ dữ liệu nào):

Nắm rõ về ManagementForm:

Chúng ta có thể đã nhận thấy phần dữ liệu bổ sung (**form-TOTAL_FORMS**, **form-INITIAL_FORMS**) được yêu cầu trong dữ liệu của Formset ở trên. Dữ liệu này là bắt buộc cho **ManagementForm**. Form này được sử dụng bởi **Formset** để quản lý tập hợp các **Form** có trong **Formset**. Nếu ta không cung cấp dữ liệu quản lý này, **Formset** sẽ không hợp lệ:

```
BaiVietFormSet = formset_factory(BaiVietForm)
data = {
    'form-0-tieu_de': 'Test',
    'form-0-pub_date': '1904-06-16',
}
formset = BaiVietFormSet(data)
formset.is_valid()
# False
```

Thuộc tính trên được sử dụng để theo dõi có bao nhiêu form instance đang được hiển thị. Nếu thêm các form mới thông qua JavaScript, chúng ta cũng nên tăng số lượng các trường trong form này. Mặt khác, nếu sử dụng JavaScript để cho phép xóa các đối tượng hiện có, thì ta cần đảm bảo các đối tượng đang bị xóa được đánh dấu thích hợp để xóa bằng cách thêm vào **form-#-DELETE** trong POST data.

Xác thực formset tùy chỉnh:

Formset có phương thức `clean` tương tự như một phương thức có trong Form class. Đây là nơi chúng ta xác định việc xác thực ở cấp bộ Formset:

```

class BaseArticleFormSet(BaseFormSet):
    def clean(self):
        """Kiểm tra xem có cặp bài viết nào bị trùng tiêu đề hay
không."""
        if any(self.errors):
            return
        ds_tieu_de = []
        for form in self.forms:
            if self.can_delete and self._should_delete_form(form):
                continue
            tieu_de = form.cleaned_data.get('tieu_de')
            if tieu_de in ds_tieu_de:
                raise ValidationError("Tiêu đề bài viết phải riê
n g biệt.")
            ds_tieu_de.append(tieu_de)

```

```

BaiVietFormSet = formset_factory(BaiVietForm, formset=BaseBaiViet
FormSet)
data = {
    'form-TOTAL_FORMS': '2',
    'form-INITIAL_FORMS': '0',
    'form-0-tieu_de': 'Test',
    'form-0-ngay_xuat_ban': '1904-06-16',
    'form-1-tieu_de': 'Test 2',
    'form-1-ngay_xuat_ban': '1912-06-23',
}
formset = BaiVietFormSet(data)
formset.is_valid()
# False
formset.errors
# [{}, {}]
formset.non_form_errors()
# ['Tiêu đề bài viết phải riêng biệt.']

```

Phương thức **clean** của **Formet** được gọi sau khi tất cả các phương thức **Form.clean** đã được gọi. Các lỗi sẽ được tìm thấy bằng cách sử dụng phương thức **non_form_errors()** trên **formset**.

Xác thực số lượng form trong một formset:

Django cung cấp một số cách để xác nhận số lượng tối thiểu hoặc tối đa các form đã gửi. Các ứng dụng cần tùy chỉnh nhiều hơn về việc xác thực số lượng form nên sử dụng xác thực formset tùy chỉnh.

- **validate_max:** Nếu **validate_max = True** được truyền vào **formet_factory()**, quá trình xác thực cũng sẽ kiểm tra xem số lượng form trong tập dữ liệu, trừ đi những form được đánh dấu để xóa và nhỏ hơn hoặc bằng **max_num**. **validate_max = True**

xác thực đúng với **max_num** ngay cả khi **max_num** bị vượt quá vì lượng dữ liệu ban đầu được cung cấp quá nhiều.

```
BaiVietFormSet = formset_factory(BaiVietForm, max_num=1, validate_max=True)
data = {
    'form-TOTAL_FORMS': '2',
    'form-INITIAL_FORMS': '0',
    'form-0-tieu_de': 'Test',
    'form-0-ngay_xuat_ban': '1904-06-16',
    'form-1-tieu_de': 'Test 2',
    'form-1-ngay_xuat_ban': '1912-06-23',
}
formset = BaiVietFormSet(data)
formset.is_valid()
# False
formset.errors
# [{}, {}]
formset.non_form_errors()
# ['Please submit at most 1 form.']
```

- Lưu ý: Bất kể giá trị của **validate_max**, nếu số lượng form trong tập dữ liệu vượt quá giá trị **absolute_max**, thì form đó sẽ không thể xác thực cho dù **validate_max** đã được set và ngoài ra, chỉ những form **absolute_max** đầu tiên sẽ được xác thực. Phần còn lại sẽ bị cắt bỏ hoàn toàn. Điều này là để bảo vệ khỏi các cuộc tấn công **memory exhaustion** (cạn kiệt bộ nhớ) bằng cách sử dụng các POST request giả mạo.

- **validate_min:** nếu **validate_min = True** được truyền vào **formset_factory()**, quá trình xác thực cũng sẽ kiểm tra xem số lượng form trong tập dữ liệu, trừ đi những form được đánh dấu để xóa và lớn hơn hoặc bằng **min_num**.

```
BaiVietFormSet = formset_factory(BaiVietForm, min_num=3, validate_min=True)
data = {
    'form-TOTAL_FORMS': '2',
    'form-INITIAL_FORMS': '0',
    'form-0-tieu_de': 'Test',
    'form-0-ngay_xuat_ban': '1904-06-16',
    'form-1-tieu_de': 'Test 2',
    'form-1-ngay_xuat_ban': '1912-06-23',
}
formset = BaiVietFormSet(data)
formset.is_valid()
# False
formset.errors
# [{}, {}]
formset.non_form_errors()
# ['Please submit at most 3 forms.']
```

Xử lý việc sắp xếp và xóa form:

- **can_order:**
 - Default: False.
 - Thuộc tính này thêm một trường bổ sung vào mỗi form. Trường mới này được đặt tên là **ORDER** và là một **forms.IntegerField**. Đối với các form đến từ dữ liệu ban đầu, nó sẽ tự động gán cho chúng một giá trị số.
- **can_delete:**

- Default: False
- Tương tự như `can_order`, thuộc tính này thêm một trường mới vào mỗi form có tên là **DELETE** và là một **forms.BooleanField**. Khi dữ liệu xuất hiện thông qua việc đánh dấu bất kỳ trường nào trong số các trường xóa, chúng ta có thể truy cập chúng bằng `delete_forms`.

Thêm các trường bổ sung vào formset:

Nếu chúng ta cần thêm các trường bổ sung vào **Formset**, điều này có thể dễ dàng hoàn thành. Formset base class cung cấp một phương thức **add_fields**. Ta có thể ghi đè phương thức này để thêm các trường của riêng mình hoặc thậm chí xác định lại các fields/attributes của order và deletion fields:

```
class BaseBaiVietFormSet(BaiVietForm):
    def add_fields(self, form, index):
        super().add_fields(form, index)
        form.fields["my_field"] = forms.CharField()
```

```
BaiVietFormSet = formset_factory(BaiVietForm, formset=BaseBaiVietFormSet)
formset = BaiVietFormSet()
for form in formset:
    print(form.as_table())
```

Truyền các thông số tùy chỉnh cho các formset forms:

Ví dụ:

```

class BaiVietCuaToiForm(BaiVietForm):
    def __init__(self, *args, user, **kwargs):
        self.user = user
        super().__init__(*args, **kwargs)

BaiVietFormSet = formset_factory(BaiVietCuaToiForm)
formset = BaiVietFormSet(form_kwargs={'user': request.user})

```

form_kwargs cũng có thể phụ thuộc vào form instance cụ thể. Formset base class cung cấp một phương thức **get_form_kwargs**. Phương thức này nhận một đối số duy nhất - chỉ mục của form trong **Formset**. Chỉ mục là **None** có đối với form rỗng

```

class BaseBaiVietFormSet(BaseFormSet):
    def get_from_kwargs(self, index):
        kwargs = super().get_from_kwargs(index)
        kwargs['custom_kwargs'] = index
        return kwargs

```

Sử dụng formset trong view trong và template:

Sử dụng một formset bên trong một view không khác lắm so với việc sử dụng một Form class thông thường. Điều duy nhất chúng ta cần lưu ý là đảm bảo sử dụng management form bên trong template. Ví dụ:

Đường dẫn templates: **app_name/templates/template_name.html**)

```
def quanly_baiviet(request):
    BaiVietFormSet = formset_factory(BaiVietForm)
    if request.method == 'POST':
        formset = BaiVietFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # Form valid, có thể thực hiện tùy ý
            pass
    else:
        formset = BaiVietFormSet()
    return render(request, 'quanly_baiviet.html', {'formset': formset})
```

Tại **quanly_baiviet.html**:

```
<form method="post">
    {{ formset.management_form }}
    <table>
        {% for form in formset %}
            {{ form }}
        {% endfor %}
    </table>
</form>
```

Sử dụng nhiều hơn một formset trong một dạng view:

Chúng ta có thể sử dụng nhiều hơn một formset trong một view nếu muốn. Formset sở hữu rất nhiều đặc tính giống với form. Chúng ta hoàn toàn có thể sử dụng tiền tố (prefix) để đặt tiền tố cho tên trường form của formset với một giá trị nhất định để cho phép nhiều hơn một formset được gửi đến một view mà không có xung đột tên. Ví dụ:

```

def quanly_baiviet(request):
    BaiVietFormSet = formset_factory(BaiVietForm)
    BinhLuanFormSet = formset_factory(BinhLuanForm)

    if request.method == 'POST':
        baiviet_formset = BaiVietFormSet(request.POST, request.FILES, prefix='bai viet')
        binhluan_formset = BinhLuanFormSet(request.POST, request.FILES, prefix='binh luan')
        if baiviet_formset.is_valid() and binhluan_formset.is_valid():
            # Xử lí form data
            pass
        else:
            baiviet_formset = BaiVietFormSet(prefix='bai viet')
            binhluan_formset = BinhLuanFormSet(prefix='binh luan')
            return render(request, 'quanly_baiviet.html',
                {'baiviet_formset': baiviet_formset,
                 'binhluan_formset': binhluan_formset})

```

Sau đó, chúng ta sẽ hiển thị các formset như bình thường. Điều quan trọng là chúng ta cần phải chuyển tiền tố cho cả trường hợp **POST** và **None-POST** request để nó được hiển thị và xử lý chính xác.

Tiền tố của mỗi formset thay thế tiền tố form mặc định được thêm vào tên của mỗi trường và thuộc tính id HTML.

2.4.4 Tạo form từ các model

2.4.4.1. Model form

Ví dụ: Chúng ta có model `BinhLuan` và ta muốn tạo một form cho phép mọi người gửi bình luận. Trong trường hợp này, sẽ là thừa nếu chúng ta tiến hành định nghĩa lại các trường đã có trong model.

Ví dụ bằng cách sử dụng class `ModelForm`:

```
class BaiVietForm(forms.Form):  
    class Meta:  
        model = BaiViet  
        field = ['tieu_de', 'noi_dung', 'ngay_xuat_ban']
```

Tạo một form để thêm bài viết

```
form = BaiVietForm()
```

Tạo một form để thay đổi bài viết sẵn có

```
baiviet = BaiViet.objects.get(pk=1)
```

```
form = BaiVietForm(instance=baiviet)
```

2.4.4.1.1. Field types

Form class được tạo sẽ có một form field cho mỗi model field được chỉ định và theo thứ tự được chỉ định trong thuộc tính của các trường.

Mỗi model field có một form field mặc định tương ứng. Ví dụ, một **CharField** trên một mô hình được biểu diễn dưới dạng **CharField** trên một **form**. Một model có **ManyToManyField** được biểu diễn dưới dạng **MultipleChoiceField**. Bên dưới là danh sách đầy đủ các chuyển đổi.

Bảng 2. 2 Danh sách chuyển đổi giữa model field và form field

Model field	Form field
-------------	------------

AutoField	Không được hiển thị trong form
BigAutoField	Không được hiển thị trong form
BigIntegerField	IntegerField với min_value được đặt thành -9223372036854775808 và max_value được đặt thành 9223372036854775807.
BinaryField	CharField , nếu muốn chỉnh sửa đặt thành True trên model field, nếu False thì không được hiển thị trong form.
BooleanField	BooleanField hoặc NullBooleanField nếu null=True .
CharField	CharField với max_length được đặt thành max_length của model field và empty_value được đặt thành None nếu null = True .
DateField	DateField
DateTimeField	DateTimeField
DecimalField	DecimalField
DurationField	DurationField
EmailField	EmailField
FileField	FileField
FilePathField	FilePathField
FloatField	FloatField
ForeignKey	ModelChoiceField
ImageField	ImageField
IntegerField	IntegerField
IPAddressField	IPAddressField
GenericIPAddressField	GenericIPAddressField
JSONField	JSONField
ManyToManyField	ModelMultipleChoiceField
NullBooleanField	NullBooleanField
PositiveBigIntegerField	IntegerField

PositiveIntegerField	IntegerField
PositiveSmallIntegerField	IntegerField
SlugField	SlugField
SmallAutoField	Không được hiển thị trong form
SmallIntegerField	IntegerField
TextField	CharField với widget=forms.Textarea
TimeField	TimeField
URLField	URLField
UUIDField	UUIDField

Các loại model field như ForeignKey và ManyToManyField là các trường hợp đặc biệt:

- ForeignKey được đại diện bởi **django.forms.ModelChoiceField**, là một **ChoiceField** có các lựa chọn là một **model QuerySet**.
- ManyToManyField được đại diện bởi **django.forms.ModelMultipleChoiceField**, là một **MultipleChoiceField** có các lựa chọn là một **model QuerySet**.

Ngoài ra, mỗi form field được tạo có các thuộc tính như sau:

- Nếu model field có **blank = True**, thì **required** được đặt sẽ được đặt thành False trên form field. Nếu không, **required = True**.
- Nhãn của form field được đặt thành **verbose_name** của trường mô hình, với ký tự đầu tiên được viết hoa.
- **Help_text** của trường form field được đặt thành **help_text** của model field.
- Nếu model field có trường **Choice** được đặt, thì tiện ích con của trường biểu mẫu sẽ được đặt thành **Select**, với các lựa chọn đến từ các lựa chọn của model field. Các lựa chọn thường sẽ bao gồm cả lựa chọn trống được chọn theo mặc định. Nếu trường này là bắt buộc, người dùng phải thực hiện lựa chọn. Lựa chọn trống sẽ không được bao gồm nếu model field có **blank = False** và một giá trị mặc định rõ ràng (giá trị mặc định ban đầu sẽ được chọn thay thế).

2.4.4.1.2. Ví dụ cụ thể

Models.py:

```
GENDER_CHOICES = [
    ('NAM', 'Nam'),
    ('NỮ', 'Nữ'),
]

class TacGia(models.Model):
    ten = models.CharField(max_length=100)
    gioitinh = models.CharField(max_length=3, choices=GENDER_CHOICES)
    ngay_sinh = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.ten

class Sach(models.Model):
    ten = models.CharField(max_length=100)
    tacgia = models.ManyToManyField(TacGia)

class TacGiaForm(ModelForm):
    class Meta:
        model = TacGia
        fields = ['ten', 'gioitinh', 'ngay_sinh']
```

Form.py

```

class TacGiaForm(ModelForm):
    class Meta:
        model = TacGia
        fields = ['ten', 'gioitinh', 'ngay_sinh']

class SachForm(ModelForm):
    class Meta:
        model = Sach
        fields = ['ten', 'tacgia']

```

2.4.4.1.3. Xác thực trên ModelForm

Có hai bước chính liên quan đến việc xác thực ModelForm:

- Xác thực form
- Xác thực model instance

Cũng giống như việc xác thực form thông thường, xác thực model form được kích hoạt ngầm khi gọi **is_valid()** hoặc truy cập thuộc tính **errors** và khi gọi **full_clean()**.

Xác thực mode (**Model.full_clean ()**) được kích hoạt bên trong bước xác thực form, ngay sau khi phương thức **clean()** của biểu mẫu được gọi.

Ghi đề phương thức clean():

- Chúng ta có thể ghi đề phương thức **clean()** trên model form để cung cấp xác nhận bổ sung giống như cách mà ta có thể làm trên form bình thường.
- Một model instance được gắn với một đối tượng model sẽ chứa một thuộc tính cung cấp cho các phương thức của nó quyền truy cập vào cá model instance cụ thể.

2.4.4.1.4. Phương thức save()

Mọi ModelForm đều có phương thức **save()**. Phương thức này tạo và lưu một đối tượng cơ sở dữ liệu từ dữ liệu được liên kết với form. Một lớp con của **ModelForm** có thể chấp nhận một model instance hiện có làm đối số từ khóa; Nếu đối số này được cung cấp, **save()** sẽ

cập nhật instance đó. Nếu đối số không được cung cấp, **save()** sẽ tạo một instance mới của model được chỉ định:

```
# Tạo một form instance từ POST data
f = BaiVietForm(request.POST)

# Lưu bài viết mới từ dữ liệu của form
baiviet = f.save()

# Tạo một form sửa đổi bài viết đã tồn tại
# Nhưng sử dụng POST data để điền vào form
a = BaiViet.objects.filter(pk=1)
f = BaiVietForm(request.POST, instance=a)
f.save()
```

Lưu ý rằng nếu form chưa được xác thực, thì việc gọi **save()** sẽ thực hiện bằng cách kiểm tra **form.errors**. Lỗi **ValueError** sẽ đưa lên nếu dữ liệu trong form không xác thực - tức là nếu **form.errors** bằng **True**.

Phương thức **save()** chấp nhận một **commit** đối số từ khóa tùy chọn, chấp nhận **True** hoặc **False**. Nếu chúng ta gọi **save()** với **commit = False**, thì nó sẽ trả về một đối tượng chưa được lưu vào cơ sở dữ liệu. Trong trường hợp này, ta có thể gọi **save()** trên model instance để lưu kết quả. Điều này sẽ hữu ích nếu ta muốn thực hiện xử lý tùy chỉnh trên đối tượng trước khi lưu đối tượng. Commit bằng True theo mặc định.

Một tác dụng phụ khác của việc sử dụng **commit = False** được thấy khi model của chúng ta có mối quan hệ nhiều-nhiều (many-to-many) với một model khác. Nếu model của chúng ta có quan hệ nhiều-nhiều và chỉ định **commit = False** khi lưu form, Django không thể lưu ngay form data cho quan hệ nhiều-nhiều. Nguyên nhân là do không thể lưu *many-to-many* data cho một instance cho đến khi instance đó tồn tại trong cơ sở dữ liệu.

Để khắc phục, mỗi khi chúng ta lưu form với **commit = False**, Django sẽ thêm phương thức **save_m2m()** vào lớp con của **ModelForm**. Sau khi ta đã lưu instance do form tạo theo cách thủ công, ta có thể gọi **save_m2m()** để lưu dữ liệu many-to-many form data. Ví dụ:

```
# Tạo một form instance từ POST data
f = BaiVietForm(request.POST)

# Tạo nhưng không save instance của bài viết mới
baiviet = f.save(commit=False)

# Sửa đổi bài viết
baiviet.tieu_de = 'TIEU_DE_MOI'

# Lưu instance mới
baiviet.save()

# Ngay lúc này tiến hành Lưu dữ liệu many-to-many
f.save_m2m()
```

Lưu ý: Chỉ cần gọi **save_m2m()** nếu sử dụng **save(commit = False)**. Khi sử dụng **save()** trên một form, tất cả dữ liệu - bao gồm many-to-many data - được lưu mà không cần gọi thêm bất kỳ phương thức nào. Ví dụ:

```
b = BaiViet(tieu_de='Hello world')
f = BaiVietForm(request.POST, instance=b)

# Tạo và Lưu instance bài viết
f.save()
```

Chọn các trường để sử dụng:

Chúng ta nên đặt rõ ràng tất cả các trường cần được chỉnh sửa trong form bằng cách sử dụng thuộc tính `fields`. Nếu không làm như vậy có thể dễ dàng dẫn đến các vấn đề bảo mật khi form bất ngờ cho phép người dùng ghi lại các trường nhất định, đặc biệt khi các trường mới được thêm vào model. Tùy thuộc vào cách form được hiển thị, vấn đề thậm chí có thể không hiển thị trên trang web.

Tuy nhiên chúng ta có thể áp dụng những cách sau nếu các vấn đề security không bị ảnh hưởng:

- Đặt thuộc tính bên trong `fields` thành “`__all__`” để chỉ ra rằng tất cả các trường trong model sẽ được sử dụng. Ví dụ:

```
class BaiVietForm(forms.Form):  
    class Meta:  
        model = BaiViet  
        fields = ['__all__']
```

- Đặt thuộc tính `exclude` của lớp `Meta` của `ModelForm` để loại trừ các trường không hiển thị ra khỏi form:

```
class BaiVietForm(forms.Form):  
    class Meta:  
        model = BaiViet  
        exclude = ['tieu_de']
```

Lưu ý:

- Bất kỳ trường nào không được bao gồm trong form theo logic trên sẽ không được lưu bởi phương thức `save()` của form. Ngoài ra, nếu chúng ta thêm các trường bị loại trừ trở lại form theo cách thủ công, chúng sẽ không được khởi tạo khi instance được lưu vào model.

- Django sẽ ngăn chặn bất kỳ nỗ lực nào cố gắng lưu một model chưa hoàn chỉnh, vì vậy nếu model không cho phép các trường trống bị thiếu và không cung cấp giá trị mặc định cho các trường bị thiếu, thì bất kỳ nỗ lực nào để **save()** một **ModelForm** có các trường bị thiếu sẽ không thành công. Để tránh lỗi này, chúng ta phải khởi tạo model của mình với các giá trị ban đầu cho các trường bị thiếu nhưng bắt buộc phải có các trường yêu cầu:

```
b = BaiViet(tieu_de='Hello world')
f = BaiVietForm(request.POST, instance=b)
f.save()
```

- Ngoài ra, chúng ta có thể sử dụng **save(commit=False)** và khởi tạo thủ công bất kỳ trường bổ sung nào.

2.4.4.1.5. Ghi đè các trường mặc định

Để chỉ định custom widget cho một field, hãy sử dụng thuộc tính widgets bên trong Meta class. Bên trong bắt buộc sử dụng dictionary.

Ví dụ: nếu chúng ta muốn CharField cho thuộc tính tiêu đề của bài viết được đặt bằng `<textarea>` thay vì `<input type = "text">` mặc định, thì ta có thể ghi đè lên field widget:

```
class BaiVietForm(forms.Form):
    slug = CharField(validators=[validate_slug])

    class Meta:
        model = BaiViet
        fields = ['tieu_de', 'noi_dung', 'ngay_xuat_ban']
        widgets = {
            'tieu_de': Textarea(attrs={'cols':80, 'rows':20})
        }
```

Tương tự, chúng ta có thể chỉ định các thuộc tính **labels**, **help_texts** và **error_messages** bên trong Meta class nếu ta muốn tiếp tục tùy chỉnh các trường. Ví dụ:

```
class BaiVietForm(forms.Form):
    slug = CharField(validators=[validate_slug])

    class Meta:
        model = BaiViet
        fields = ['tieu_de', 'noi_dung', 'ngay_xuat_ban']
        labels = {
            'tieu_de': _('Tieu de'),
        }
        help_texts = {
            'tieu_de': _('Text bất kì.'),
        }
        error_messages = {
            'tieu_de': {
                'max_length': _("Tiêu đề quá dài."),
            }
        }
```

Cuối cùng, nếu chúng ta muốn kiểm soát hoàn toàn một trường - bao gồm **type**, **validators**, **required** v.v. - ta có thể thực hiện việc này bằng cách chỉ định các trường rồi khai báo giống như cách chúng ta làm trong Form thông thường.

Nếu chúng ta muốn chỉ định phần xác thực của một trường, ta có thể làm như vậy bằng cách xác định trường, khai báo và đặt tham số **validators**:


```

class BaiVietForm(forms.Form):
    slug = CharField(validators=[validate_slug])

    class Meta:
        model = BaiViet
        fields = ['tieu_de', 'noi_dung', 'ngay_xuat_ban']

```

Lưu ý: Các trường được định nghĩa một cách khai báo sẽ không sử dụng lại các ràng buộc mà trường đó đã có trong model như **max_length** hoặc **required** từ model tương ứng. Nếu chúng ta muốn giữ những ràng buộc đã có trong model, ta phải đặt các đối số liên quan một cách rõ ràng khi khai báo form field.

Ví dụ chúng ta có model BaiViet như sau:

```

class BaiViet(models.Model):
    tieu_de = models.CharField(
        max_length=200,
        null=True,
        blank=True,
        help_text='Some help text'
    )
    ngay_khoi_tao = models.DateTimeField(auto_now=True)
    noi_dung = models.TextField()

```

Và chúng ta muốn thực hiện một số tùy chỉnh cho tiêu đề, trong khi vẫn giữ giá trị **blank** và giá trị **help_text** như đã chỉ định trong model, ta có thể xác định BaiVietForm như sau:

```

class BaiVietForm(forms.Form):
    tieu_de = CharField(
        max_length=200,
        required=False,
        help_text='Some help text'
    )

    class Meta:
        model = BaiViet
        fields = ['tieu_de', 'noi_dung', 'ngay_xuat_ban']

```

Cho phép localization (bản địa hóa) các trường:

- Hệ thống **format** của Django có khả năng hiển thị ngày, giờ và số trong các template sử dụng định dạng được chỉ định cho ngôn ngữ hiện tại. Nó cũng xử lý đầu vào được bản địa hóa trong các form.
- Khi tính năng này được bật, hai người dùng truy cập cùng một nội dung có thể thấy ngày, giờ và số được định dạng theo những cách khác nhau, tùy thuộc vào định dạng cho ngôn ngữ hiện tại của họ.
- Theo mặc định, các trường trong **ModelForm** sẽ không bản địa hóa (localize) dữ liệu. Để cho phép bản địa hóa cho các trường, ta có thể sử dụng thuộc tính **localized_fields** trong **Meta class**.

```

class BaiVietForm(forms.Form):
    class Meta:
        model = BaiViet
        fields = ['tieu_de', 'noi_dung', 'ngay_xuat_ban']
        localized_fields = ('ngay_xuat_ban', )

```

- Nếu **localized_fields** được đặt thành giá trị “__all__”, thì tất cả các trường sẽ được bản địa hóa.

2.4.4.1.6. Kế thừa Form

Với Django chúng ta hoàn toàn có thể mở rộng và sử dụng lại **ModelForms** bằng cách kế thừa chúng. Ví dụ: sử dụng lớp **BaiVietForm** trước đó

```
class NCBaiVietForm(BaiVietForm):  
    def clean_noi_dung(self):  
        pass
```

Điều này tạo ra một form hoạt động giống như **BaiVietForm**, ngoại trừ việc có thêm một số thay đổi cho trường “**noi_dung**”.

Chúng ta cũng có thể thay đổi bên trong Meta class nếu muốn thay đổi **Meta.fields** hoặc **Meta.exclude**:

```
class NCBaiVietForm(BaiVietForm):  
    def clean_noi_dung(self):  
        pass  
  
class ChuyenHuongBaiVietForm(NCBaiVietForm):  
    class Meta(BaiVietForm.Meta):  
        exclude = ('ngay_xuat_ban', )
```

Thực hiện thay đổi bằng cách thêm phương thức bổ sung từ **NCBaiVietForm** và sửa đổi **BaiVietForm.Meta** ban đầu để xóa trường “**ngay_xuat_ban**”.

Tuy nhiên, có một số điều cần lưu ý:

- Nếu chúng ta có nhiều lớp cơ sở khai báo Meta inner class, thì chỉ lớp đầu tiên sẽ được sử dụng. Điều này có nghĩa là các Meta child class, nếu tồn tại thì sẽ được sử dụng. Ngược lại sử dụng Meta class của parent đầu tiên.
- Có thể kế thừa đồng thời từ cả Form và ModelForm, tuy nhiên, phải đảm bảo rằng ModelForm xuất hiện đầu tiên trong MRO (Method Resolution Order - xác định đường dẫn tìm kiếm lớp được Python sử dụng để tìm kiếm phương thức phù hợp để sử dụng trong các lớp có đa kế thừa). Điều này là do các lớp này phụ thuộc vào metaclass khác nhau và một lớp chỉ có thể có một metaclass.

Cung cấp giá trị ban đầu:

Giống như với các form thông thường, có thể chỉ định dữ liệu khởi tạo ban đầu cho các form bằng cách chỉ định một tham số ban đầu khi khởi tạo form. Các giá trị khởi tạo ban đầu sẽ ghi đè cả giá trị ban đầu từ form field và các giá trị từ một model instance đính kèm.

Ví dụ:

```
baiviet = BaiViet.objects.get(pk=1)
print(baiviet.tieu_de)
# Hello world

form = BaiVietForm(initial={'tieu_de': 'Covid-
19'}, instance=BaiViet)
form['tieu_de'].value()
# Covid-19
```

2.4.4.2. Model formsets

Tương tự như các formset thông thường, Django cung cấp một vài lớp cải tiến formset để làm việc với Django form thuận tiện hơn:

```
BaiVietFormSet = modelformset_factory(BaiViet, fields=('tieu_de',
'noi_dung'))
```

2.4.4.2.1. Thay đổi queryset

Theo mặc định, khi chúng ta tạo formset từ một model, formset sẽ sử dụng **queryset** bao gồm tất cả các đối tượng trong model (ví dụ: **BaiViet.objects.all()**). Chúng ta có thể ghi đè bằng cách sử dụng đối số cho **queryset**:

```
formset = modelformset_factory(  
    queryset=BaiViet.objects.filter(tieu_de__startswith='O'))
```

2.4.4.2.2. Lưu các đối tượng trong formset

Tương tự với **ModelForm**, chúng ta có thể lưu dữ liệu dưới dạng một model object. Điều này được thực hiện bằng phương thức **save()** của **formset**:

```
# Tạo formset instance với POST data.  
formset = BaiVietFormSet(request.POST)  
  
# Giả sử dữ liệu hợp lệ, tiến hành Lưu dữ liệu  
instances = formset.save()
```

Phương thức **save()** trả về các instance đã được lưu vào cơ sở dữ liệu. Nếu dữ liệu của một instance nhất định không thay đổi trong dữ liệu liên kết, thì instance đó sẽ không được lưu vào cơ sở dữ liệu và sẽ không được bao gồm trong giá trị trả về.

Thiết lập **commit = False** để trả về các model instance chưa được lưu:

```
# không Lưu dữ liệu vào database  
instances = formset.save(commit=False)  
  
for instance in instances:  
    instance.save()
```

Điều này cung cấp cho chúng ta khả năng đính kèm dữ liệu vào các instance trước khi lưu chúng vào cơ sở dữ liệu. Nếu formset chứa **ManyToManyField**, ta cũng sẽ cần gọi

formet.save_m2m() để đảm bảo mối quan hệ **many-to-many** (nhiều-nhiều) được lưu đúng cách.

Sau khi gọi **save()**, **Model formset** của chúng ta sẽ có ba thuộc tính mới chứa các thay đổi của formset:

- **models.BaseModelFormSet.changed_objects**
- **models.BaseModelFormSet.deleted_objects**
- **models.BaseModelFormSet.new_objects**

2.4.4.2.3. Giới hạn số lượng đối tượng có thể hiển thị

Tương tự như với các formset thông thường, chúng ta có thể sử dụng **max_num** và tham số **extra** cho **modelformset_factory()** để giới hạn số lượng các form bổ sung được hiển thị.

Lưu ý: **max_num** không ngăn các đối tượng hiện có được hiển thị:

```
BaiViet.objects.order_by('tieu_de')
# <QuerySet [<BaiViet: Hello world>, <BaiViet: Covid - 19>, <BaiViet: The Dawg>]>

BaiVietFormSet = modelformset_factory(BaiViet, fields=('tieu_de',
), max_num=1)
formset = BaiVietFormSet(queryset=BaiViet.objects.order_by('tieu_de'))
[x.tieu_de for x in formset.get_queryset()]
# ['Hello world', 'Covid - 19', 'The Dawg']
```

Ngoài ra, **extra = 0** không ngăn cản việc tạo các model instance mới vì chúng ta có thể thêm các form bổ sung bằng JavaScript hoặc gửi thêm POST data. Formset chưa cung cấp chức năng cho chế độ “**edit only**” view để ngăn cản việc tạo các instance mới.

Nếu giá trị của **max_num** lớn hơn số lượng các đối tượng hiện có, thì các form trống sẽ được thêm bổ sung vào **formset**, miễn là tổng số biểu mẫu không vượt quá **max_num**.

2.4.4.2.4. Sử dụng Model formset trong một view

Model formset rất giống với formset. Giả sử chúng ta muốn hiển thị một formset để chỉnh sửa các BaiViet model instance:

```
def quanly_baiviet(request):
    BaiVietFormSet = modelformset_factory(BaiViet, fields=('tieu_de', 'noi_dung'))
    if request.method == 'POST':
        formset = BaiVietFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # do something.
    else:
        formset = BaiVietFormSet()
    return render(request, 'quanly_baiviet.html', {'formset': formset})
```

2.4.4.2.5. Tùy chỉnh queryset

Chúng ta có thể ghi đè queryset mặc định của model formset như sau:

```

def quanly_baiviet(request):
    tac_gia = TacGia.objects.get(pk=1)
    BaiVietFormSet = inlineformset_factory(BaiViet, TacGia, fields
=('tieu_de', 'noi_dung'))
    if request.method == 'POST':
        formset = BaiVietFormSet(request.POST, request.FILES
                                ,instance=tac_gia)

        if formset.is_valid():
            formset.save()
            # Điều hướng khi thực hiện thành công
            return HttpResponseRedirect(tac_gia.get_absolute_url(
))
        else:
            formset = BaiVietFormSet()
            return render(request, 'quanly_baiviet.html', {'formset': for
mset})

```

2.4.4.2.6. Sử dụng Formset trong một template

Có ba cách để hiển thị một formset trong một Django template.

Đầu tiên, chúng ta có thể để formset thực hiện hầu hết các công việc:

```

<form method="post">
    {{ formset }}
</form>

```

Thứ hai, chúng ta có thể render formset theo cách thủ công. Lưu ý đảm bảo sử dụng render management form như bên dưới:


```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
</form>
```

Thứ ba, ta có thể hiển thị từng trường theo cách thủ công như sau:

```
<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {% for field in form %}
            {{ field.label_tag }} {{ field }}
        {% endfor %}
    {% endfor %}
</form>
```

Nếu lựa chọn sử dụng phương pháp thứ ba và không lặp lại các trường thông qua **{% for %}**, chúng ta sẽ cần render trường khóa chính. Ví dụ: nếu chúng ta đang cần render các trường tiêu đề và nội dung của một model:

```

<form method="post">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form.id }}
        <ul>
            <li>{{ form.tieu_de }}</li>
            <li>{{ form.noi_dung }}</li>
        </ul>
    {% endfor %}
</form>

```

2.4.4.3. Inline formsets

Inline formsets là một lớp trừu tượng nhỏ nằm trên các model formset. Inline formsets giúp đơn giản hóa trong trường hợp làm việc với các đối tượng liên quan thông qua khóa ngoại. Giả sử chúng ta có hai model sau:

```

class TacGia(models.Model):
    ten = models.CharField(max_length=100)
    tuoi = models.IntegerField()
    gioitinh = models.CharField(max_length=3,
choices=GENDER_CHOICES)

class BaiViet(models.Model):
    tieu_de = models.CharField(max_length=200)
    ngay_khoi_tao = models.DateTimeField(auto_now=True)
    noi_dung = models.TextField()
    tac_gia = models.ForeignKey(TacGia, on_delete=models.CASCADE)

```

Nếu chúng ta muốn tạo một formset cho phép chỉnh sửa bài viết của một tác giả cụ thể, ta có thể làm như sau:

```

from django.forms import inlineformset_factory
BaiVietFormSet = inlineformset_factory(TacGia, BaiViet, fields=('
tieu_de',))
tac_gia = TacGia.objects.get(ten='Tuan Tran Quoc')
formset = BaiVietFormSet(instance=tac_gia)

```

Trường hợp nhiều khóa ngoại trên cùng một model:

Nếu model chứa nhiều hơn một khóa ngoại, ta sẽ cần giải quyết sự không rõ ràng theo cách thủ công bằng cách sử dụng **fk_name**. Ví dụ, xét model sau:

```

class Friendship(models.Model):
    from_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name='from_friends',
    )
    to_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name='friends',
    )
    point = models.IntegerField()

```

Để giải quyết vấn đề này, ta có thể sử dụng **fk_name** trong **inlineformset_factory()**:

```

FriendshipFormSet = inlineformset_factory(
    Friend, Friendship, fk_name='from_friend',
    fields=('to_friend', 'point'))

```

Sử dụng inline formset trong một view:

Chúng ta có thể cung cấp một view cho phép người dùng chỉnh sửa các đối tượng liên quan của một model. Ví dụ:

```
def quanly_baiviet(request):
    tac_gia = TacGia.objects.get(pk=1)
    BaiVietFormSet = inlineformset_factory(BaiViet,
TacGia, fields=('tieu_de', 'noi_dung'))
    if request.method == 'POST':
        formset = BaiVietFormSet(request.POST, request.FILES
                                ,instance=tac_gia)

        if formset.is_valid():
            formset.save()
            # Điều hướng khi thực hiện thành công
            return HttpResponseRedirect(tac_gia.get_abso-
lute_url())
        else:
            formset = BaiVietFormSet()
            return render(re-
quest, 'quanly_baiviet.html', {'formset': formset})
```

2.4.5. Form Assets (Media class)

Ứng dụng Django Admin xác định một số tiện ích tùy chỉnh cho calendars, filtered selections, v.v. Các tiện ích con này xác định các yêu cầu về nội dung và Django Admin sử dụng chúng này để tùy chỉnh thay cho Django mặc định. Admin templates sẽ chỉ bao gồm những tệp được yêu cầu để hiển thị tiện ích con trên bất kỳ trang nào.

Nếu chúng ta cân nhắc việc sử dụng các tiện ích mà Django Admin sử dụng, hãy thoải mái sử dụng chúng trong ứng dụng Tất cả chúng đều được lưu trữ trong **django.contrib.admin.widgets**.

2.4.5.1. Nội dung dưới dạng định nghĩa static

Cách dễ nhất để xác định nội dung là định nghĩa static. Để sử dụng phương thức này, ta có khai báo là bên trong một Media class. Các thuộc tính của lớp bên trong xác định các yêu cầu.

Ví dụ:

```
from django import forms

class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

Đoạn code trên xác định một **CalendarWidget** dựa trên **TextInput**. Mỗi khi **CalendarWidget** được sử dụng trên một form, form đó sẽ được chuyển hướng để bao gồm tệp CSS **pretty.css** và các tệp JavaScript **animations.js** và **action.js**.

Định nghĩa static này được chuyển đổi trong thời gian chạy thành một thuộc tính tiện ích con có tên là **media**. Có thể truy xuất danh sách các nội dung cho một **CalendarWidget** instance thông qua thuộc tính này:

```
w = CalendarWidget()
print(w.media)
# <link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet">
# <script src="http://static.example.com/animations.js"></script>
# <script src="http://static.example.com/actions.js"></script>
```

2.4.5.2. CSS

Chúng ta cần một dictionary để mô tả các tệp CSS cần thiết cho các media output.

Các giá trị trong dictionary phải là **tuple/danh sách tên tệp**. Xem phần về đường dẫn để biết chi tiết về cách chỉ định đường dẫn đến các tệp này.

Các key trong dictionary là loại media output. Đây là những loại tương tự được các tệp CSS chấp nhận trong khai báo media: 'all', 'aural', 'braille', 'embossed', 'handheld', 'print', 'projection', 'screen', 'tty' and 'tv'. Nếu chúng ta cần có các biểu định kiểu khác nhau cho các loại media khác nhau, cung cấp danh sách các tệp CSS cho từng output media. Ví dụ sau sẽ cung cấp hai tùy chọn CSS - một cho screen và một cho print:

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'print': ('newspaper.css',)
    }
```

Kết quả sau khi render ra HTML như sau:

```
<link href="http://static.example.com/pretty.css" type="text/css"
media="screen" rel="stylesheet">
<link href="http://static.example.com/lo_res.css" type="text/css"
media="tv,projector" rel="stylesheet">
<link href="http://static.example.com/newspaper.css" type="text/c
ss" media="print" rel="stylesheet">
```

2.4.5.3. Media như một thuộc tính động:

Nếu chúng ta cần thực hiện một số thao tác phức tạp hơn đối với các yêu cầu nội dung, ta có thể xác định thuộc tính media trực tiếp. Điều này được thực hiện bằng cách xác định một thuộc tính widget trả về một thể hiện của các forms.Media. Hàm tạo cho các biểu

forms.Media chấp nhận các đối số từ khóa css và js ở cùng một định dạng được sử dụng trong định nghĩa media tĩnh.

Ví dụ: định nghĩa tĩnh cho Calendar Widget cũng có thể được định nghĩa theo kiểu động:

```
class CalendarWidget(forms.TextInput):
    @property
    def media(self):
        return forms.Media(css={'all': ('pretty.css',)},
                           js=('animations.js', 'actions.js'))
```

2.4.5.4. Đường dẫn của các file nội dung:

Đường dẫn được sử dụng để chỉ định nội dung có thể là tương đối hoặc tuyệt đối. Nếu một đường dẫn bắt đầu bằng /, **http: //** hoặc **https: //**, nó sẽ được hiểu là một đường dẫn tuyệt đối và được giữ nguyên. Tất cả các đường dẫn khác sẽ được thêm vào trước giá trị của tiền tố thích hợp. Nếu **django.contrib.staticfiles** được cài đặt, ứng dụng này sẽ được sử dụng để phân phát nội dung.

Cho dù chúng ta có thiết lập sử dụng **django.contrib.staticfiles** hay không, thì cài đặt **STATIC_URL** và **STATIC_ROOT** là bắt buộc để hiển thị một trang web hoàn chỉnh.

Để tìm tiền tố thích hợp để sử dụng, Django sẽ kiểm tra xem nếu cài đặt **STATIC_URL** không phải **None** và tự động quay lại sử dụng **MEDIA_URL**. Ví dụ: nếu **MEDIA_URL** cho trang web là: *'http://uploads.example.com/'* và **STATIC_URL** là **None**:

```

from django import forms
class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('/css/pretty.css',),}

        js = ('animations.js', 'http://othersite.com/actions.js')

w = CalendarWidget()
print(w.media)
# <link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet">
# <script src="http://uploads.example.com/animations.js"></script>
# <script src="http://othersite.com/actions.js"></script>

```

Nhưng nếu **STATIC_URL** được thiết lập là `'http://static.example.com/'`:

```

w = CalendarWidget()
print(w.media)
# <link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet">
# <script src="http://static.example.com/animations.js"></script>
# <script src="http://othersite.com/actions.js"></script>

```

Hoặc nếu **staticfiles** được định cấu hình bằng **ManifestStaticFilesStorage**:


```
w = CalendarWidget()
print(w.media)
# <link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet">
# <script src="https://static.example.com/animations.27e20196a850.js"></script>
# <script src="http://othersite.com/actions.js"></script>
```

2.4.5.5. Đối tượng Media

Khi yêu cầu thuộc tính media của một widget hoặc form, giá trị được trả về là một đối tượng forms.Media. Như chúng ta đã thấy, chuỗi biểu diễn của một đối tượng Media là HTML cần thiết để bao gồm các tệp có liên quan trong khối <head> của trang HTML.

Tuy nhiên, các đối tượng Media có một số thuộc tính thú vị khác.

Tập hợp nội dung con:

Nếu chúng ta chỉ muốn các tệp thuộc một loại cụ thể, ta có thể sử dụng tham số bên dưới để lọc ra. Ví dụ:

```
w = CalendarWidget()
print(w.media)
# <link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet">
# <script src="http://static.example.com/animations.js"></script>
# <script src="http://static.example.com/actions.js"></script>

print(w.media['css'])
# <link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet">
```

Kết hợp các đối tượng Media:

Các đối tượng media cũng có thể được thêm vào cùng nhau. Khi hai đối tượng Media được thêm vào, kết quả đối tượng Media chứa liên hợp các nội dung được chỉ định bởi cả hai:

```
from django import forms

class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')

class OtherWidget(forms.TextInput):
    class Media:
        js = ('whizbang.js',)

w1 = CalendarWidget()
w2 = OtherWidget()
print(w1.media + w2.media)
# <link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet">
# <script src="http://static.example.com/animations.js"></script>
# <script src="http://static.example.com/actions.js"></script>
# <script src="http://static.example.com/whizbang.js"></script>
```

Thứ tự nội dung:

Thứ tự mà nội dung được chèn vào DOM thường rất quan trọng. Ví dụ: chúng ta có thể có một tập lệnh phụ thuộc vào jQuery. Do đó, việc kết hợp các đối tượng Media cố gắng duy trì thứ tự tương đối trong đó các nội dung được xác định trong mỗi lớp Media.

```

from django import forms
class CalendarWidget(forms.TextInput):
    class Media:
        js = ('jQuery.js', 'calendar.js', 'noConflict.js')
class TimeWidget(forms.TextInput):
    class Media:
        js = ('jQuery.js', 'time.js', 'noConflict.js')
w1 = CalendarWidget()
w2 = TimeWidget()
print(w1.media + w2.media)
# <script src="http://static.example.com/jquery.js"></script>
# <script src="http://static.example.com/calendar.js"></script>
# <script src="http://static.example.com/time.js"></script>
# <script src="http://static.example.com/noConflict.js"></script>

```

Media trên form:

Các widget không phải là đối tượng duy nhất có thể có các định nghĩa **media** - các form cũng có thể xác định **media**. Các quy tắc cho định nghĩa media trên form cũng giống như các quy tắc cho tiện ích con: khai báo có thể là tĩnh hoặc động; đường dẫn và quy tắc kế thừa cho các khai báo đó hoàn toàn giống nhau.

Bất kể chúng ta có xác định khai báo **media** hay không, tất cả các đối tượng Form đều có thuộc tính **media**. Giá trị mặc định cho thuộc tính này là kết quả của việc thêm các định nghĩa **media** cho tất cả các tiện ích con là một phần của form:

```
from django import forms
class ContactForm(forms.Form):
    date = DateField(widget=CalendarWidget)
    name = CharField(max_length=40, widget=OtherWidget)

f = ContactForm()
f.media
# <link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet">
# <script src="http://static.example.com/animations.js"></script>
# <script src="http://static.example.com/actions.js"></script>
# <script src="http://static.example.com/whizbang.js"></script>
```

Nếu chúng ta muốn liên kết các nội dung bổ sung với một form- ví dụ: CSS cho bố cục form - thêm khai báo **Media** vào form:

```

class ContactForm(forms.Form):
    date = DateField(widget=CalendarWidget)
    name = CharField(max_length=40, widget=OtherWidget)

class Media:
    css = {
        'all': ('layout.css',)
    }

f = ContactForm()
f.media
# <link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet">
# <link href="http://static.example.com/layout.css" type="text/css" media="all" rel="stylesheet">
# <script src="http://static.example.com/animations.js"></script>
# <script src="http://static.example.com/actions.js"></script>
# <script src="http://static.example.com/whizbang.js"></script>

```

2.5. Template

2.6. Class-based views

Class-based views cung cấp một cách thay thế để triển khai các views dưới dạng các đối tượng Python thay vì các function. Chúng không thay thế các view dựa trên function, nhưng có những khác biệt và lợi thế nhất định khi so sánh với các view dựa trên function:

- Tổ chức mã liên quan đến các phương thức HTTP cụ thể (GET, POST, v.v.) có thể được giải quyết bằng các phương pháp riêng biệt thay vì phân nhánh có điều kiện.
- Các kỹ thuật hướng đối tượng như mixin (đa kế thừa) có thể được sử dụng để phân tích mã thành các thành phần có thể tái sử dụng.

2.6.1. Sử dụng Class-based views

Về cốt lõi, Class-based views cho phép phản hồi các phương thức HTTP khác nhau bằng các phương thức instance class khác nhau, thay vì với mã phân nhánh có điều kiện bên trong một view function.

Vì vậy, nơi mã để xử lý HTTP GET trong một view function sẽ giống như sau:

```
from django.http import HttpResponseRedirect

def my_view(request):
    if request.method == 'GET':
        # <view logic>
        return HttpResponseRedirect('result')
```

Trong Class-based views, đoạn mã trên sẽ tương ứng với:

```
from django.http import HttpResponseRedirect
from django.views import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponseRedirect('result')
```

Tại urls.py:

```
# urls.py
from django.urls import path
from myapp.views import MyView

urlpatterns = [
    path('about/', MyView.as_view()),
]
```

Với Class-based views chúng ta hoàn toàn có thể ghi đè các thuộc tính của lớp cha:

```
from django.http import HttpResponse
from django.views import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

Ghi đè tại lớp con:

```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

2.6.2. Xử lý forms với class-based views

Một view dựa trên function cơ bản xử lý các form có thể trông giống như sau:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import MyForm

def myview(request):
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')
    else:
        form = MyForm(initial={'key': 'value'})

    return render(request, 'form_template.html', {'form': form})
```

Tương tự, trong class-based view:


```

from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views import View

from .forms import MyForm

class MyFormView(View):
    form_class = MyForm
    initial = {'key': 'value'}
    template_name = 'form_template.html'

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {'form': form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')

        return render(request, self.template_name, {'form': form})

```

2.6.3. Decorate pattern trong class

Một phương thức trên một class không hoàn toàn giống với một function độc lập, vì vậy chúng ta không thể chỉ áp dụng một decorate function method - trước tiên ta cần phải chuyển đổi nó thành một decorate method. Decorate **method_decorator** biến đổi một

decorate function thành một decorate method để nó có thể được sử dụng trên một instance method. Ví dụ:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

class ProtectedView(TemplateView):
    template_name = 'secret.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
```

Hoặc ngắn gọn hơn, chúng ta có thể decorate class thay thế và chuyển tên của phương thức cần decorate như là đối số từ khóa **name**:

```
@method_decorator(login_required, name='dispatch')
class ProtectedView(TemplateView):
    template_name = 'secret.html'
```

Trong các ví dụ trên, mọi phiên bản của **ProtectedView** sẽ có bảo vệ đăng nhập.

2.6.4. Xây dựng trong class-based generic views

2.6.4.1. Generic views của các đối tượng

TemplateView chắc chắn hữu ích, nhưng Django **generic views** thực sự tỏa sáng khi thể hiện các view dựa trên nội dung cơ sở dữ liệu. Django đi kèm với một số **generic views** được tích hợp sẵn để giúp tạo chế độ xem danh sách và chi tiết của các đối tượng.

Ví dụ, chúng ta có các model sau:

```

# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __str__(self):
        return self.name

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)

```

Tiếp đến, định nghĩa phần view:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherListView(ListView):
    model = Publisher
```

Cuối cùng nối phần view vào url:

```
# urls.py
from django.urls import path
from books.views import PublisherListView

urlpatterns = [
    path('publishers/', PublisherListView.as_view()),
]
```

2.7. Migrations

Migrations là cách Django truyền bá những thay đổi mà ta thực hiện đối với model của mình (thêm trường, xóa model, v.v.) vào giản đồ cơ sở dữ liệu. Chúng được thiết kế để chủ yếu là tự động, nhưng chúng ta sẽ cần biết khi nào thực hiện migrations, khi nào chạy chúng và các vấn đề phổ biến mà ta có thể gặp phải.

2.7.1. Commands

Có một số lệnh mà chúng ta sẽ sử dụng để tương tác với quá trình migrations và việc xử lý lược đồ cơ sở dữ liệu của Django:

- **migrate**, nơi chịu trách nhiệm áp dụng và không áp dụng migrations.
- **makemigrations**, chịu trách nhiệm tạo migrations mới dựa trên những thay đổi ta đã thực hiện đối với model của mình.

- **sqlmigrate**, hiển thị các câu lệnh SQL để migrations.
- **showmigrations**, liệt kê danh sách các migrations của project và trạng thái của chúng.

Migrations giống như một hệ thống kiểm soát phiên bản cho lược đồ cơ sở dữ liệu. **make-migrations** chịu trách nhiệm đóng gói các thay đổi model của ta thành các tệp migration - tương tự như các commits - và **migrate** chịu trách nhiệm áp dụng các thay đổi đó vào cơ sở dữ liệu.

Các tệp migration cho mỗi ứng dụng nằm trong thư mục “migrations” bên trong ứng dụng và được thiết kế để commit và phân phối như một phần của cơ sở mã của ứng dụng.

Migrations sẽ chạy theo cùng một cách trên cùng một tập dữ liệu và tạo ra kết quả nhất quán, có nghĩa là những gì ta thấy trong quá trình phát triển, trong cùng một trường hợp, chính xác là những gì sẽ xảy ra trong quá trình production.

Django sẽ thực hiện migrations đối với bất kỳ thay đổi nào đối với model hoặc trường của - ngay cả các tùy chọn không ảnh hưởng đến cơ sở dữ liệu - vì cách duy nhất nó có thể tạo lại trường một cách chính xác là có tất cả các thay đổi trong lịch sử và ta có thể cần các tùy chọn đó trong một số migrations sau này.

2.7.2. Workflow

Django có thể tạo migrations. Bằng cách thực hiện các thay đổi đối với model - giả sử, thêm một trường hay xóa một model - sau đó chạy **makemigrations**:

```
$ python manage.py makemigrations
Migrations for 'books':
  books/migrations/0003_auto.py:
    - Alter field author on book
```

Các model sẽ được quét và so sánh với các phiên bản hiện có trong tệp migrations và sau đó một tập hợp migrations mới sẽ được viết ra. Hãy đảm bảo đọc kết quả đầu ra để xem **makemigrations** nghĩ chúng ta đã thay đổi những gì. Đôi lúc, đối với những thay đổi phức

tập, **makemigrations** có thể không phát hiện ra những thay đổi mà ta đã thực hiện, hãy lưu ý và kiểm tra lại.

Sau khi có các tệp migrations mới, ta nên áp dụng chúng vào cơ sở dữ liệu của mình để đảm bảo chúng hoạt động như mong đợi:

```
$ python manage.py migrate
```

Operations to perform:

```
  Apply all migrations: books
```

Running migrations:

```
  Rendering model states... DONE
```

```
  Applying books.0003_auto... OK
```

Khi quá trình migrations được áp dụng, commits migrations và các thay đổi đối với hệ thống kiểm soát phiên bản (version control) bằng một lần commit - theo cách đó, khi các nhà phát triển khác (hoặc server) kiểm tra mã, họ sẽ nhận được cả hai thay đổi đối model và migrations.

Version control:

Vì quá trình migrations có thể xảy ra cùng một lúc nên đôi khi sẽ xảy ra trường hợp cả hai developer đều tạo ra **migrations** mới tại cùng một thời điểm. Tuy nhiên chúng ta hoàn toàn có thể kiểm soát được vấn đề này bằng cách kiểm tra migrations được tạo với **migrations** liên hệ trước đó, thông qua đó có thể tìm được sự đụng độ giữa các **migrations**.

2.7.3. Transactions

Trên những cơ sở dữ liệu hỗ trợ **DDL transactions** (SQLite và PostgreSQL), tất cả các hoạt động **migrations** sẽ chạy bên trong một transactions theo mặc định. Ngược lại, nếu cơ sở dữ liệu không hỗ trợ các giao dịch DDL (ví dụ: MySQL, Oracle) thì tất cả các hoạt động sẽ chạy mà không có **transactions**.

2.7.4. Dependencies

Mặc dù quá trình **migrations** diễn ra theo từng ứng dụng, nhưng các bảng và mối quan hệ giữa các model là quá phức tạp để có thể tạo cho một ứng dụng tại một thời điểm. Khi chúng ta thực hiện quá trình **migrations** yêu cầu một ràng buộc khác để chạy - ví dụ: chúng ta thêm **ForeignKey** trong ứng dụng Book vào ứng dụng Author- kết quả **migrations** sẽ phụ thuộc vào việc **migrations** của ứng dụng Author.

Điều này có nghĩa là khi chúng ta chạy **migrations**, quá trình **migrations** Author sẽ chạy trước và tạo ra bảng tham chiếu **ForeignKey**, sau đó quá trình **migrations** tạo ra ràng buộc **ForeignKey**.

Các ứng dụng không có **migrations** không được có quan hệ (**ForeignKey**, **ManyToManyField**, v.v.) với các ứng dụng có **migrations**.

2.7.5. Migrations file

Quá trình **migrations** được lưu trữ dưới dạng “**migration files**”. Các tệp này là các tệp Python bình thường với bố cục đối tượng được thống nhất, được viết theo kiểu khai báo.

Tệp migration cơ bản trông giống như sau:

```
from django.db import migrations, models

class Migration(migrations.Migration):

    dependencies = [
        ('migrations', '0001_initial')]

    operations = [
        migrations.DeleteModel('Tribble'),
        migrations.AddField('Author', 'rating', models.IntegerField(
            default=0)),
    ]
```

Django sẽ tìm kiếm một lớp con của **django.db.migrations.Migration** được gọi là **Migration** khi load migrations file. Sau đó, nó kiểm tra đối tượng này để tìm bốn thuộc tính, chỉ hai trong số đó được sử dụng thường xuyên:

- **dependencies**: Danh sách các migrations phụ thuộc.
- **operations**, danh sách các hoạt động mà file migrations này sẽ thực hiện

operations chính là chìa khóa; chúng là một tập hợp các hướng dẫn khai báo cho Django biết những thay đổi nào cần được thực hiện.

2.7.5. Initial migrations

“**initial migrations**” cho một ứng dụng là những **migrations** tạo ra phiên bản đầu tiên của các bảng của ứng dụng đó. Thông thường một ứng dụng sẽ có một **initial migrations**, nhưng trong một số trường hợp phụ thuộc lẫn nhau của các model phức tạp, có thể có hai hoặc nhiều hơn.

initial migrations được đánh dấu bằng thuộc tính lớp **initial = True** trên lớp **migrations**. Nếu không tìm thấy thuộc tính của lớp **initial**, một **migrations** sẽ được coi là “**initial**” nếu đó là lần **migration** đầu tiên trong ứng dụng (tức là nếu nó không có phụ thuộc vào bất kỳ lần **migration** nào khác trong cùng một ứng dụng).

Khi tùy chọn **migrate --fake-initial** được sử dụng, những **initial migrations** này được xử lý đặc biệt. Đối với lần **migration** đầu tiên tạo ra một hoặc nhiều bảng (CreateModel), Django kiểm tra xem tất cả các bảng đó đã tồn tại trong cơ sở dữ liệu chưa và giả mạo áp dụng **migration** nếu có. Tương tự, đối với lần **migration** ban đầu có thêm một hoặc nhiều trường (AddField), Django kiểm tra xem tất cả các cột tương ứng đã tồn tại trong cơ sở dữ liệu và giả mạo áp dụng quá trình **migration** nếu có. Nếu không có **--fake-initial**, những lần **migration** ban đầu được xem xét không khác với bất kỳ **migration** khác.

2.7.6. Thêm migration vào ứng dụng

Các ứng dụng mới được định cấu hình sẵn để chấp nhận **migration** và vì vậy chúng ta có thể thêm **migration** bằng cách chạy chỉnh sửa sau khi đã thực hiện một số thay đổi.

Nếu ứng dụng của chúng ta đã có các model và bảng cơ sở dữ liệu và chưa có **migration** (cơ sở dữ liệu và model có sẵn), ta có thể thực hiện áp dụng migration bằng cách sau:

```
$ python manage.py makemigrations your_app_label
```


Điều này sẽ áp dụng **initial migrations** vào app. Bây giờ, hãy chạy **python management.py migrate --fake-initial** và Django sẽ quét, kiểm tra và phát hiện chúng ta có một initial migration và các bảng đã tồn tại sau đó đánh dấu việc **migration** là đã được áp dụng.

Lưu ý rằng điều này chỉ hoạt động đáp ứng hai điều kiện sau:

- Chúng ta không thay đổi model kể từ khi bảng được tạo từ trước đó. Để quá trình **migration** hoạt động, trước tiên ta phải thực hiện quá trình **initial migrations** ban đầu và sau đó thực hiện các thay đổi, vì Django so sánh các thay đổi với các tệp **migration**, không phải cơ sở dữ liệu.
- Chúng ta chưa chỉnh sửa cơ sở dữ liệu của mình theo cách thủ công - Django sẽ không thể phát hiện ra rằng cơ sở dữ liệu không khớp với các model, ta sẽ chỉ gặp lỗi khi **migration** cố gắng sửa đổi các bảng đó.

2.7.7. Đảo ngược quá trình migration

Quá trình **migration** có thể được đảo ngược bằng cách chuyển số đến lần migration trước đó. Ví dụ: Để đảo ngược migrations **books.0003**:

```
...\> py manage.py migrate books zero
Operations to perform:
  Unapply all migrations: books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0002_auto... OK
  Unapplying books.0001_initial... OK
```

2.8. Quản lí File

2.8.1. Sử dụng tệp trong các model

Khi chúng ta sử dụng **FileField** hoặc **ImageField**, Django cung cấp một bộ API để ta có thể sử dụng để xử lý tệp.

Xét model sau, sử dụng **ImageField** để lưu trữ ảnh:

```
from django.db import models
```

```
class Car(models.Model):  
    name = models.CharField(max_length=255)  
    price = models.DecimalField(max_digits=5, decimal_places=2)  
    photo = models.ImageField(upload_to='cars')
```

Bất kỳ Car instance nào cũng sẽ có thuộc tính photo và ta có thể sử dụng để lấy thông tin chi tiết của ảnh:

```
>>> car = Car.objects.get(name="57 Chevy")  
>>> car.photo  
<ImageFieldFile: cars/chevy.jpg>  
>>> car.photo.name  
'cars/chevy.jpg'  
>>> car.photo.path  
'/media/cars/chevy.jpg'  
>>> car.photo.url  
'http://media.example.com/cars/chevy.jpg'
```

2.8.2. Đối tượng tệp

Django sử dụng **django.core.files.File** bất cứ lúc nào nó cần để đại diện cho một tệp.

Ví dụ:

```
>>> from django.core.files import File  
  
# Tạo python file bằng cách sử dụng open()  
>>> f = open('/path/to/hello.world', 'w')  
>>> myfile = File(f)
```

Lưu ý rằng các tệp được tạo theo cách này không tự động đóng. Phương pháp sau có thể được sử dụng để đóng tệp :

```
>>> from django.core.files import File

# Tạo python file sử dụng open()
>>> with open('/path/to/hello.world', 'w') as f:
...     myfile = File(f)
...     myfile.write('Hello World')
...
>>> myfile.closed
True
>>> f.closed
True
```

2.8.3. File storage

Tích hợp lớp lưu trữ hệ thống tệp tin:

Django cung cấp lớp **django.core.files.storage.FileSystemStorage**, lớp này thực hiện lưu trữ tệp hệ thống tệp cục bộ.

Ví dụ: Đoạn mã sau sẽ lưu trữ các tệp đã tải lên trong **/media/photos** bất kể cài đặt **MEDIA_ROOT** là gì:

```

from django.core.files.storage import FileSystemStorage
from django.db import models

fs = FileSystemStorage(location='/media/photos')

class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)

```

2.9. Signals (Tín hiệu)

2.9.1. Lắng nghe signals

Để nhận tín hiệu, hãy đăng ký một hàm **receiver** bằng phương thức **Signal.connect()**. Chức năng thu được gọi khi tín hiệu được gửi đi. Tất cả các hàm receiver của tín hiệu được gọi lần lượt theo thứ tự chúng đã được đăng ký.

Signal.connect(receiver, sender=None, weak=True, dispatch_uid=None):

Parameters:

- **receiver** – Hàm callback sẽ được kết nối với **signal**.
- **sender** - Chỉ định một người gửi cụ thể để nhận **signal**.
- **weak** - Django lưu trữ trình xử lý tín hiệu dưới dạng tham chiếu yếu theo mặc định. Do đó, nếu **receiver** của chúng ta là một hàm cục bộ, nó có thể bị thu gom rác. Để tránh điều này, truyền vào **weak = False** khi gọi phương thức signal's **connect()**.
- **dispatch_uid** - Một mã định danh duy nhất cho **receiver signal** trong trường hợp các tín hiệu trùng lặp có thể được gửi đi.

2.9.1.1. Hàm receiver

Đầu tiên, chúng ta cần xác định một hàm receiver. receiver có thể là bất kỳ hàm hoặc phương thức Python nào:

```
def my_callback(sender, **kwargs):  
    print("Request finished!")
```

2.9.1.2. Kết nối các hàm receiver

Có hai cách chúng có thể kết nối receiver. Ta có thể sử dụng kết nối thủ công vào route:

```
from django.core.signals import request_finished  
request_finished.connect(my_callback)
```

Ngoài ra, chúng có thể sử dụng **receiver()** decorator:

```
from django.core.signals import request_finished  
from django.dispatch import receiver  
  
@receiver(request_finished)  
def my_callback(sender, **kwargs):  
    print("Request finished!")
```

Ngay lúc này, hàm **my_callback** của chúng ta sẽ được gọi mỗi khi một request kết thúc.

2.9.1.3. Kết nối đến tín hiệu được gửi bởi những sender cụ thể

Những signal thường được sử dụng:

- **django.db.models.signals.pre_save**: Tín hiệu sẽ được gửi ngay ở đầu phương thức **save()** của model (model chưa thực sự được save vào database).
- **django.db.models.signals.post_save**: Giống với **pre_save** nhưng tín hiệu được gửi ở cuối phương thức **save()** của model (model đã được save vào database).

Ví dụ: Để đăng kí nhận tín hiệu từ model **MyModel** ta có thể làm như sau:

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from myapp.models import MyModel

@receiver(post_save, sender=MyModel)
def my_handler(sender, **kwargs):
    ...
```

Hàm **my_handler** sẽ chỉ được gọi khi một phiên bản của **MyModel** được lưu.

2.10. User Authentication

2.10.1. Sử dụng hệ thống xác thực Django

2.10.1.1. Đối tượng User

Đối tượng người dùng là cốt lõi của hệ thống xác thực. Chúng thường đại diện cho những người tương tác với trang web. Chỉ một lớp user tồn tại trong **Django’s authentication framework**. ”superusers” hoặc ”admin” quản trị viên chỉ là những user với các thuộc tính đặc biệt được thiết lập, không phải là các lớp khác nhau của các đối tượng User.

The primary attributes of the default user are:

Các thuộc tính chính của user mặc định là:

- **username**
- **password**
- **email**
- **first_name**
- **last_name**

2.10.1.2. Tạo user

Cách trực tiếp nhất để tạo user là sử dụng hàm trợ giúp **create_user()** được bao gồm:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')

# User đã được save vào database
# Ta có thể tiến hành thay đổi các field của user
# Hoặc chúng ta có thể add thêm field mới
>>> user.last_name = 'Lennon'
>>> user.save()
```

2.10.1.3. Tạo superusers

Tạo superusers bằng lệnh **createsuperuser**:

```
$ python manage.py createsuperuser --username=joe --
email=joe@example.com
```

2.10.1.4. Thay đổi password

Django không lưu trữ raw password trên model user, mà chỉ lưu trữ giá trị hash (băm).

manage.py changepassword *username* là phương pháp sử dụng dòng lệnh để thay đổi user password.

Chúng ta cũng có thể thay đổi password bằng cách sử dụng `set_password()`:

```
from django.contrib.auth.models import User
u = User.objects.get(username='john')
u.set_password('new password')
u.save()
```

Hoặc chúng ta còn có thể sử dụng admin page để thay đổi password.

2.10.1.5. Chứng thực user

Để chứng thực một user chúng ta có thể sử dụng **authenticate()** cho tập hợp của các thông tin xác thực. Phương thức này trả về đối tượng user nếu tập hợp các thông tin xác thực là chính xác. Ngược lại raise **PermissionDenied** và trả về None. Ví dụ:

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    # Chứng thực thành công
else:
    # Chứng thực thất bại
```

2.10.1.6. Permissions và Authorization

Django đi kèm với một hệ thống permissions được tích hợp sẵn. Hệ thống cung cấp cách để gán quyền cho người dùng và nhóm người dùng cụ thể.

Trang web Django admin sử dụng các permissions như sau:

- Quyền truy cập để xem các đối tượng bị giới hạn ở những người dùng có quyền “view” (xem) hoặc “change” (thay đổi) đối với loại đối tượng đó.
- Quyền truy cập để “add” (thêm) form và thêm đối tượng bị giới hạn đối với người dùng có quyền “add” cho loại đối tượng đó.
- Quyền truy cập để xem danh sách thay đổi, xem “change” form và thay đổi đối tượng được giới hạn cho người dùng có quyền “change” đối với loại đối tượng đó.
- Quyền truy cập để xóa một đối tượng bị giới hạn ở những người dùng có quyền “delete” đối với loại đối tượng đó.

10.1.7. Chứng thực trong Web request:

Django sử dụng sessions và middleware để kết nối hệ thống xác thực vào các đối tượng yêu cầu.

Django cung cấp thuộc tính **request.user** trên mọi request đại diện cho user hiện tại. Nếu người dùng hiện tại chưa đăng nhập, thuộc tính này sẽ được đặt thành một instance của **AnonymousUser**, ngược lại thuộc tính này sẽ được đặt thành một instance của **User**.

Chúng ta có thể sử dụng **is_authenticated** như sau:

```
if request.user.is_authenticated:
    # User qua bước này đã chứng thực
    ...
else:
    # User qua bước này chưa chứng thực
    ...
```

2.10.1.6.1. Đăng nhập

Chúng ta có thể sử dụng hàm **login()** thuộc **django.contrib.auth.login()** để đăng nhập user.

Ví dụ bên dưới sử dụng đồng thời **authenticate()** và **login()** để hoàn thành quá trình xác thực:

```

from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=username, password=password)
    if user is not None:
        login(request, user)
        # Login thành công, tiến hành chuyển hướng.
        ...
    else:
        # Trả về 'invalid login' error message.
        ...

```

2.10.1.6.2. Đăng xuất

Chúng ta có thể sử dụng hàm **logout()** thuộc **django.contrib.auth.logout()** để đăng xuất user.

Ví dụ:

```

from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Tiến hành chuyển hướng khi Logout.

```

2.10.1.6.3. Sử dụng chứng thực trong template

Lưu ý trước khi sử dụng form POST action để chứng thực, luôn luôn sử dụng **csrf_token**.

Ví dụ:

```

<form method="post" action="{% url 'login' %}">
    {% csrf_token %}
    <table>
        <tr>
            <td>{{ form.username.label_tag }}</td>
            <td>{{ form.username }}</td>
        </tr>
        <tr>
            <td>{{ form.password.label_tag }}</td>
            <td>{{ form.password }}</td>
        </tr>
    </table>
    <input type="submit" value="login">
    <input type="hidden" name="next" value="{{ next }}">
</form>

```

Ngoài ra, chúng ta hoàn toàn có thể sử dụng phương thức **request.user.is_authenticated()** để kiểm tra chứng thực trước khi hiển thị nội dung:

```

{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
    <p>Welcome, new user. Please log in.</p>
{% endif %}

```

Đối với permissions, ta có thể sử dụng phương thức **perm.<app_name>.<permission_name>** để tiến hành kiểm tra permission:

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.add_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.add_driving %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

2.10.2. Quản lý password trong Django

2.10.2.1. Lưu trữ password

Django cung cấp một hệ thống lưu trữ mật khẩu linh hoạt và sử dụng **PBKDF2** theo mặc định.

Thuộc tính password của user là một string có định dạng như sau:

```
<algorithm>${<iterations>}${<salt>}${<hash>}
```

Các thành phần của password được phân tách bởi dấu \$ và bao gồm: Tên thuật toán hash, số lần lặp của thuật toán hash, random salt và kết quả hash.

Theo mặc định, Django sử dụng thuật toán PBKDF2 với hàm băm SHA256, một cơ chế kéo dài mật khẩu được **NIST** khuyến nghị. Khá an toàn và đòi hỏi một lượng lớn thời gian tính toán để phá vỡ.

Mặc định PASSWORD_HASHERS là:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
]
```

2.10.2.2. *Bật xác thực mật khẩu*

Xác thực mật khẩu được định cấu hình trong cài đặt:

AUTH_PASSWORD_VALIDATORS:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
        'OPTIONS': {
            'min_length': 9,
        }
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]
```

Ví dụ trên cho phép tất cả bốn trình xác thực được bao gồm:

- **UserAttributeSimilarityValidator**, kiểm tra sự giống nhau giữa mật khẩu và một tập hợp các thuộc tính của người dùng.

- **MinimumLengthValidator**, kiểm tra xem mật khẩu có đáp ứng độ dài tối thiểu hay không. Trình xác thực này cấu hình với tùy chọn tùy chỉnh: yêu cầu độ dài tối thiểu là chín ký tự thay vì tám ký tự mặc định.
- **CommonPasswordValidator**, kiểm tra xem mật khẩu có nằm trong danh sách các mật khẩu phổ biến hay không. Theo mặc định, nó được so sánh với một danh sách bao gồm 20.000 mật khẩu phổ biến.
- **NumericPasswordValidator**, kiểm tra xem mật khẩu có phải hoàn toàn là số hay không.

2.11. Serializing objects

2.11.1. Serializing data

Ở mức cao nhất, chúng ta có thể **serialize** dữ liệu như sau:

```
from django.core import serializers
data = serializers.serialize("xml", SomeModel.objects.all())
```

Các đối số của hàm **serialize** là định dạng để **serialize** dữ liệu và một **QuerySet** để **serialize**.

django.core.serializers.get_serializer(format)

Chúng ta cũng có thể sử dụng trực tiếp một đối tượng serializer:

```
XMLSerializer = serializers.get_serializer("xml")
xml_serializer = XMLSerializer()
xml_serializer.serialize(queryset)
data = xml_serializer.getvalue()
```

Điều này hữu ích nếu chúng ta muốn **serialize** dữ liệu trực tiếp đến một đối tượng dạng tệp (bao gồm **HttpResponse**):

```
with open("file.xml", "w") as out:
    xml_serializer.serialize(SomeModel.objects.all(), stream=out)
```

2.11.1.1. Serialize tập hợp con của các trường

Nếu chúng ta chỉ muốn một tập hợp con các trường được **serialize**, ta có thể chỉ định một đối số trường cho **serializer**:

```
from django.core import serializers
data = serializers.serialize('xml',
                             SomeModel.objects.all(), fields=('name', 'size'))
```

Trong ví dụ này, chỉ các thuộc tính **name** và **size** của mỗi model sẽ được **serialize**. Khóa chính luôn được **serialize** dưới dạng phần tử pk trong kết quả đầu ra và không bao giờ xuất hiện trong phần trường.

2.11.1.2. Serialize model được kế thừa

Nếu chúng ta có một model sử dụng kế thừa nhiều bảng, ta cũng cần phải **serialize** tất cả các lớp cơ sở cho model đó. Điều này là do chỉ các trường được xác định cục bộ trên model sẽ được **serialize**. Ví dụ, xét các model sau:

```
class Place(models.Model):
    name = models.CharField(max_length=50)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
```

Nếu chỉ **serialize** model Restaurant:

```
data = serializers.serialize('xml', Restaurant.objects.all())
```

Các trường ở output đầu ra được **serialize** sẽ chỉ chứa thuộc tính **serves_hot_dogs**. Thuộc tính **name** của **base class** sẽ bị bỏ qua.

Để **serialize** đầy đủ Restaurant instance, chúng ta cũng cần phải **serialize** các model **Place**:

```
all_objects = [*Restaurant.objects.all(), *Place.objects.all()]
data = serializers.serialize('xml', all_objects)
```

2.11.2. Deserializing data

Việc **deserializing** data rất giống với việc **serialize** data:

```
for obj in serializers.deserialize("xml", data):
    do_something_with(obj)
```

Như chúng ta có thể thấy, hàm **deserialize** nhận đối số định dạng giống như **serialize**, một chuỗi hoặc một luồng dữ liệu và trả về một bộ lặp.

Tuy nhiên, các đối tượng được trả về bởi bộ lặp **deserialize** không phải là các đối tượng Django thông thường. Thay vào đó, chúng là các đối tượng **DeserializedObject** đặc biệt bao bọc một đối tượng đã tạo - nhưng chưa được lưu - và bất kỳ dữ liệu quan hệ liên quan nào.

Gọi **DeserializedObject.save()** lưu đối tượng vào cơ sở dữ liệu.

Lưu ý: Nếu pk của đối tượng bằng null hoặc không tồn tại thì đối tượng sẽ được lưu vào cơ sở dữ liệu.

Tuy nhiên nếu không tin tưởng vào danh sách đối tượng được lưu, chúng ta có thể lưu dữ liệu bằng cách sau:

```
for deserialized_object in serializers.deserialize("xml", data):
    if object_should_be_saved(deserialized_object):
        deserialized_object.save()
```

Bản thân đối tượng **Django** có thể được kiểm tra dưới dạng **deserialized_object.object**. Nếu các trường trong dữ liệu được **serialize** không tồn tại trên model, lỗi **DeserializationError** sẽ xuất hiện và trừ khi đối số **ignorenonexistent** qua được chuyển vào là **True**:

```
serializers.deserialize("xml", data, ignorenonexistent=True)
```


2.11.3. Serialization formats

Django hỗ trợ một số định dạng **serialize**, một số định dạng yêu cầu cài đặt các mô-đun Python của bên thứ ba:

Định danh	Thông tin
xml	serializes XML đơn giản.
json	serializes đến và từ JSON.
jsonl	serializes đến và từ JSONL.
yaml	serializes đến yaml

2.12. Ký mã hóa

nguyên tắc vàng về bảo mật ứng dụng Web là không bao giờ tin cậy dữ liệu từ các nguồn không đáng tin cậy. Đôi khi, việc chuyển dữ liệu qua một phương tiện không đáng tin cậy có thể hữu ích. Các giá trị được ký bằng mật mã có thể được chuyển qua một kênh không đáng tin cậy một cách an toàn với hiểu biết rằng mọi hành vi giả mạo sẽ được phát hiện.

Django cung cấp cả API cấp thấp để ký giá trị và API cấp cao để thiết lập và đọc cookie đã ký, một trong những cách sử dụng phổ biến nhất của việc đăng nhập trong các ứng dụng Web.

2.12.1. Bảo vệ SECRET_KEY

Khi chúng tạo một Django project mới bằng startproject, tệp settings.py được tạo tự động và nhận giá trị **SECRET_KEY** ngẫu nhiên. Giá trị này là chìa khóa để bảo mật dữ liệu đã ký - điều quan trọng là ta phải giữ an toàn cho dữ liệu này, nếu không những kẻ tấn công có thể sử dụng nó để tạo các giá trị đã ký của riêng chúng.

2.12.2. Sử dụng các API cấp thấp

Các phương pháp ký của Django có trong mô-đun **django.core.signs**. Để ký một giá trị, trước tiên khởi tạo một **Signer instance**:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign('My string')
>>> value
'My string:GdMGD6HNQ_qdgyYP8yBZAdAIV1w'
```

Chữ ký được nối vào cuối chuỗi, sau dấu hai chấm. Chúng ta có thể truy xuất giá trị ban đầu bằng phương thức **unsign**:

```
>>> original = signer.unsign(value)
>>> original
'My string'
```

Nếu chúng ta chuyển một giá trị không phải là chuỗi để ký, giá trị sẽ bị buộc phải chuyển thành chuỗi trước khi được ký và kết quả **unsign** sẽ cung cấp cho chúng ta giá trị chuỗi:

```
>>> signed = signer.sign(2.5)
>>> original = signer.unsign(signed)
>>> original
'2.5'
```

Nếu muốn bảo vệ một list, bộ tuple hoặc dictionary, ta có thể sử dụng phương thức **sign_object()** và **unsign_object()**:

```
>>> signed_obj = signer.sign_object({'message': 'Hello!'})
>>> signed_obj
'eyJtZXNzYWdlIjoiaGVsbG8hIn0:Xdc-m0FDjs22KsQAqfVfi8PQSPdo3ckWJxPWwQOFhR4'
>>> obj = signer.unsign_object(signed_obj)
>>> obj
{'message': 'Hello!'}
```

Nếu chữ ký hoặc giá trị đã bị thay đổi theo bất kỳ cách nào, **django.core.signs.BadSignature** exception sẽ được đưa ra:

```
>>> from django.core import signing
>>> value += 'm'
>>> try:
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
```

Mặc định **Signer** class sử dụng **SECRET_KEY** trong setting để tạo chữ ký. Chúng ta có thể sử dụng secret khác bằng cách chuyển secret key mới này vào **Signer** constructor:

```
>>> signer = Signer('my-other-secret')
>>> value = signer.sign('My string')
>>> value
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

2.12.2.1. Sử dụng đối số *salt*

Nếu chúng ta không muốn chuỗi băm bị lặp, ta có thể sử dụng đối số **salt** tùy chọn cho lớp **Signer**. Sử dụng một **salt** sẽ tạo ra hàm băm với **salt** và **SECRET_KEY**:

```

>>> signer = Signer()
>>> signer.sign('My string')
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer.sign_object({'message': 'Hello!'})
'eyJtZXNzYWdlIjoIcGVsbG8hIn0:Xdc-
m0FDjs22KsQAqfVfi8PQSPdo3ckWJxPWwQOFhR4'
>>> signer = Signer(salt='extra')
>>> signer.sign('My string')
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign('My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw')
'My string'
>>> signer.sign_object({'message': 'Hello!'})
'eyJtZXNzYWdlIjoIcGVsbG8hIn0:-UWSLCE-
oUAHzhkHviYz3SOZYBjFKl1EOyVZNuUtM-I'
>>> signer.unsign_object('eyJtZXNzYWdlIjoIcGVsbG8hIn0:-UWSLCE-
oUAHzhkHviYz3SOZYBjFKl1EOyVZNuUtM-I')
{'message': 'Hello!'}

```

2.12.2.2. Xác minh thời gian của chữ ký

TimestampSigner là một lớp con của **Signer** đưa thời gian ký vào giá trị đã ký. Điều này cho phép ta xác nhận rằng giá trị đã ký nhận được được tạo trong một khoảng thời gian cụ thể:

```

>>> from datetime import timedelta
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign('hello')
>>> value
'hello:1NMg5H:oPVuCq1JWmChm1rA2lyTutelC-c'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
...
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value, max_age=timedelta(seconds=20))
'hello'

```

CHƯƠNG 3: TRIỂN KHAI ỨNG DỤNG

3. Tổng quan

Khi trang web của chúng ta đã hoàn thành (hoặc đã hoàn thành "đủ" để bắt đầu thử nghiệm công khai), ta sẽ cần lưu trữ nó ở nơi nào đó công khai và dễ truy cập hơn so với máy tính cá nhân.

Cho đến thời điểm hiện tại, chúng ta đang làm việc trong môi trường phát triển, sử dụng máy chủ web Django để chia sẻ trang web của ta với trình duyệt/mạng cục bộ và chạy trang web với cài đặt phát triển hiển thị gỡ lỗi (không an toàn) và các thông tin quan trọng khác. Trước khi có thể lưu trữ một trang web bên ngoài, đầu tiên chúng ta sẽ cần phải:

- Thực hiện một vài thay đổi đối với project settings.
- Chọn môi trường để lưu trữ ứng dụng Django.
- Chọn một môi trường để lưu trữ tệp static.
- Thiết lập cơ sở hạ tầng ở mức production-level để cung cấp trang web.

3.1. Môi trường production là gì?

Môi trường production là môi trường được cung cấp bởi máy chủ, nơi chúng ta sẽ chạy trang web của mình. Môi trường production bao gồm:

- Phần cứng máy tính.
- Hệ điều hành (ví dụ: Linux, Windows).
- Programming language runtime và framework libraries được sử dụng cho website.
- Máy chủ web được sử dụng để cung cấp các trang và nội dung khác (ví dụ: Nginx, Apache).
- Máy chủ ứng dụng chuyển các yêu cầu "động" giữa trang web Django của chúng ta và máy chủ web.
- Cơ sở dữ liệu mà trang web đang sử dụng.

3.2. Chọn nhà cung cấp dịch vụ lưu trữ

Có hơn 100 nhà cung cấp dịch vụ lưu trữ được biết là hỗ trợ tích cực hoặc hoạt động tốt với Django (chúng ta có thể tìm thấy danh sách khá đầy đủ tại <https://djangofriendly.com/index.html>). Các nhà cung cấp này cung cấp các loại môi trường khác nhau (IaaS, PaaS) và các cấp độ máy tính và tài nguyên mạng khác nhau với các mức giá khác nhau.

Một số điều cần xem xét khi chọn máy chủ:

- Độ lớn của trang web và chi phí dữ liệu và tài nguyên máy chủ cần thiết để đáp ứng nhu cầu đó.
- Mức độ hỗ trợ mở rộng quy mô theo chiều ngang (thêm nhiều máy hơn) và theo chiều dọc (nâng cấp lên các máy mạnh hơn) và chi phí để thực hiện việc này.
- Nơi nhà cung cấp có các trung tâm dữ liệu để đạt được tốc độ cao nhất.
- Hiệu suất thời gian hoạt động và thời gian ngừng hoạt động lịch sử của máy chủ.
- Các công cụ được cung cấp để quản lý trang web - chúng có dễ sử dụng không và chúng có bảo mật không (ví dụ: SFTP so với FTP).
- Các công cụ có sẵn để giám sát máy chủ.
- Những hạn chế: Một số máy chủ sẽ cố tình chặn một số dịch vụ nhất định (ví dụ: email). Những người khác chỉ cung cấp một số giờ "thời gian trực tiếp" nhất định trong một số mức giá hoặc chỉ cung cấp một lượng nhỏ dung lượng lưu trữ.
- Lợi ích kèm theo. Một số nhà cung cấp sẽ cung cấp tên miền miễn phí và hỗ trợ chứng chỉ SSL.
- Sử dụng cấp bậc lưu trữ "miễn phí" có hết hạn theo thời gian hay không và liệu chi phí chuyển sang cấp bậc khác có đắt hơn hay không. Điều đó khiến chúng ta nên cân nhắc việc sử dụng nhà cung cấp dịch vụ khác.

3.3. Heroku

Heroku là một trong những dịch vụ PaaS dựa trên đám mây phổ biến và lâu đời nhất. Ban đầu nó chỉ hỗ trợ các ứng dụng Ruby, nhưng bây giờ có thể được sử dụng để lưu trữ các ứng dụng từ nhiều môi trường lập trình, bao gồm cả Django!

Chúng ta chọn sử dụng Heroku vì một số lý do:

- Heroku có một cấp bậc miễn phí thực sự miễn phí (mặc dù có một số hạn chế).
- Là một PaaS, Heroku chăm sóc rất nhiều cơ sở hạ tầng web cho chúng ta. Điều này giúp ta bắt đầu dễ dàng hơn nhiều vì không phải lo lắng về máy chủ, bộ cân bằng tải, proxy ngược hoặc bất kỳ cơ sở hạ tầng web nào khác mà Heroku cung cấp cho chúng ta.
- Mặc dù nó có một số hạn chế nhưng sẽ không ảnh hưởng đến ứng dụng cụ thể. Ví dụ:
 - Heroku chỉ cung cấp dung lượng lưu trữ ngắn hạn nên các tệp do người dùng tải lên không thể được lưu trữ an toàn trên Heroku.
 - Cấp bậc miễn phí sẽ rơi vào trạng thái ngủ nếu một ứng dụng web không hoạt động hoặc không có yêu cầu nào trong khoảng thời gian nửa giờ. Sau đó, trang web có thể mất vài giây để phản hồi khi được đánh thức.
 - Cấp bậc miễn phí giới hạn thời gian trang web của chúng ta chạy trong một khoảng thời gian nhất định hàng tháng (không bao gồm thời gian trang web ở trạng thái "ngủ").
- Hầu hết đều hoạt động tốt. Nếu hoàn toàn ưng ý với nhà cung cấp này chúng ta hoàn toàn có thể scale up một cách dễ dàng.

3.3.1. Cách thức hoạt động của Heroku

Heroku chạy các trang web Django trong một hoặc nhiều **"Dynos"**, là các vùng chứa Unix được ảo hóa, biệt lập, cung cấp môi trường cần thiết để chạy một ứng dụng. Các dynos hoàn toàn bị cô lập và có một hệ thống tệp tạm thời (một hệ thống tệp tồn tại trong thời gian ngắn được làm sạch / làm trống mỗi khi dyno khởi động lại). Điều duy nhất mà dynos

chia sẻ theo mặc định là các biến cấu hình ứng dụng. Heroku nội bộ sử dụng bộ cân bằng tải để phân phối lưu lượng truy cập web cho tất cả các dynos "web". Vì không có gì được chia sẻ giữa chúng, Heroku có thể mở rộng ứng dụng theo chiều ngang bằng cách thêm nhiều dynos hơn.

Vì hệ thống tệp là tạm thời nên chúng ta không thể cài đặt trực tiếp các dịch vụ mà ứng dụng yêu cầu (ví dụ: cơ sở dữ liệu, hàng đợi, hệ thống bộ nhớ đệm, bộ nhớ, dịch vụ email, v.v.). Thay vào đó, các ứng dụng web Heroku sử dụng các dịch vụ hỗ trợ được cung cấp dưới dạng "tiện ích bổ sung" độc lập bởi Heroku hoặc các bên thứ 3. Sau khi được đính kèm vào ứng dụng web của mình, các dynos truy cập vào các dịch vụ bằng cách sử dụng thông tin có trong các biến cấu hình ứng dụng.

Để thực thi ứng dụng, Heroku cần có khả năng thiết lập môi trường và phụ thuộc thích hợp, đồng thời hiểu cách nó được khởi chạy. Đối với các ứng dụng Django, chúng tôi cung cấp thông tin này trong một số tệp văn bản:

- **runtime.txt**: ngôn ngữ lập trình và phiên bản sử dụng.
- **requirements.txt**: các phụ thuộc thành phần Python, bao gồm cả Django.
- **Procfile**: Danh sách các quy trình sẽ được thực thi để khởi động ứng dụng web. Đối với Django, đây thường sẽ là máy chủ ứng dụng web Gunicorn (với tập lệnh `.wsgi`).
- **wsgi.py**: Cấu hình WSGI để gọi ứng dụng Django của chúng tôi trong môi trường Heroku.

3.3.2. Setup các tệp tin cần thiết cho Heroku

Procfile:

Tạo tệp **Procfile** (không có phần mở rộng) trong thư mục root:

```
web: gunicorn <project_name>.wsgi --log-file -
```

"**web**:" cho Heroku biết rằng đây là một **web dyno** và có thể được gửi lưu lượng truy cập HTTP. Quá trình bắt đầu trong dyno này là gunicorn, đây là một máy chủ ứng dụng web

phổ biến mà Heroku đề xuất. Chúng ta bắt đầu Gunicorn bằng cách sử dụng thông tin cấu hình trong mô-đun `<project_name>.wsgi` (`/<project_name>/wsgi.py`).

Gunicorn:

Gunicorn là máy chủ HTTP được khuyến nghị sử dụng với Django trên Heroku (như được tham chiếu trong Procfile ở trên). Nó là một máy chủ HTTP thuần Python cho các ứng dụng **WSGI** có thể chạy nhiều quy trình Python đồng thời trong một **dyno** duy nhất:

Cài đặt Gunicorn:

```
pip install gunicorn
```

Cung cấp các tệp tĩnh trong môi trường production:

Để dễ dàng lưu trữ các tệp tĩnh riêng biệt với ứng dụng web Django, Django cung cấp công cụ **collectstatic** để thu thập các tệp này khi triển khai. Django templates yêu cầu vị trí lưu trữ của các tệp tĩnh liên quan đến một biến cài đặt (**STATIC_URL**).

Các biến cài đặt liên quan bao gồm:

- **STATIC_URL**: Đây là vị trí URL cơ sở mà từ đó các tệp tĩnh sẽ được phân phát.
- **STATIC_ROOT**: Đây là đường dẫn tuyệt đối đến một thư mục nơi công cụ **collectstatic** của Django sẽ thu thập bất kỳ tệp tĩnh nào được tham chiếu trong các template của chúng ta. Sau khi được thu thập, các tệp này sau đó có thể được tải lên thành một nhóm đến bất kỳ nơi nào các tệp sẽ được lưu trữ.
- **STATICFILES_DIRS**: Đây là danh sách liệt kê các thư mục bổ sung mà công cụ **collectstatic** của Django sẽ tìm kiếm các tệp tĩnh.

Chúng ta tiến hành thêm những đoạn code sau vào bên dưới cùng của tệp tin **settings.py**:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.9/howto/static-files/
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
STATIC_URL = '/static/'

# Extra places for collectstatic to find static files.
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'static'),
)
```

Tiếp đến tiến hành việc cài đặt Whitenoise để phục vụ nội dung tĩnh trực tiếp từ Gunicorn trong quá trình production.

Cài đặt Whitenoise:

```
pip install whitenoise
```

Để cài đặt Whitenoise vào Django ta tiến hành việc thêm Whitenoise vào **MIDDLEWARE** trong tệp tin settings.py:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
    'whitenoise.middleware.WhiteNoiseMiddleware',  
]
```

Tiếp tục ta thêm phần bên dưới vào dưới cùng của tệp tin settings.py để hoàn thành việc cài đặt Whitenoise:

```
STATICFILES_STORAGE = 'whitenoise.storages.CompressedManifestStaticFilesStorage'
```

Runtime:

Tệp **runtime.txt**, nếu được định nghĩa, sẽ cho Heroku biết ngôn ngữ lập trình nào được sử dụng. Tạo tệp trong thư mục gốc của repo và thêm vào dòng sau:

```
python-3.8.11
```

Cấu hình Database:

Chúng ta không thể sử dụng cơ sở dữ liệu SQLite mặc định trên Heroku vì nó dựa trên tệp và nó sẽ bị xóa khỏi hệ thống tệp tạm thời của Heroku mỗi khi ứng dụng khởi động lại (thường là mỗi ngày một lần và mỗi khi ứng dụng hoặc các biến cấu hình của nó bị thay đổi).

Thông tin kết nối cơ sở dữ liệu được cung cấp cho **dyno web** bằng cách sử dụng biến cấu hình có tên **DATABASE_URL**. Thay vì mã hóa cứng thông tin này thành Django, Heroku

khuyến nghị các nhà phát triển sử dụng gói **dj-database-url** để phân tích cú pháp biến môi trường **DATABASE_URL** và tự động chuyển đổi nó sang định dạng cấu hình mong muốn của Django:

```
pip3 install dj-database-url
```

Tại settings.py thêm phần cấu hình sau ở cuối file:

```
# Heroku: Update database configuration from $DATABASE_URL.  
import dj_database_url  
db_from_env = dj_database_url.config(conn_max_age=500)  
DATABASES['default'].update(db_from_env)
```

Requirement:

Các Python requirements của ứng dụng web phải được lưu trữ trong tệp tin **requirements.txt** trong thư mục root. Heroku sau đó sẽ tự động cài đặt chúng khi nó xây dựng lại môi trường. Chúng ta có thể tạo tệp này bằng cách sử dụng pip trên command line:

```
pip freeze > requirements.txt
```

3.3.3. Tạo tài khoản Heroku

Để bắt đầu sử dụng Heroku, trước tiên chúng ta cần tạo một tài khoản:

- Truy cập www.heroku.com và nhấp vào nút **SIGN UP FOR FREE**.
- Nhập thông tin chi tiết và sau đó nhấn **CREATE FREE ACCOUNT**. Chúng ta sẽ được yêu cầu kiểm tra tài khoản của mình để tìm email đăng ký.
- Nhấp vào liên kết kích hoạt tài khoản trong email đăng ký. Chúng ta sẽ được đưa trở lại tài khoản của mình trên trình duyệt web.
- Nhập mật khẩu và nhấp vào **SET PASSWORD AND LOGIN..**
- Sau đó, chúng ta sẽ đăng nhập và được đưa đến bảng điều khiển Heroku: <https://dashboard.heroku.com/apps>

3.3.4. Cài đặt Heroku client

Tải xuống và cài đặt Heroku client bằng cách làm theo hướng dẫn trên Heroku tại đây:

<https://devcenter.heroku.com/articles/getting-started-with-python#set-up>

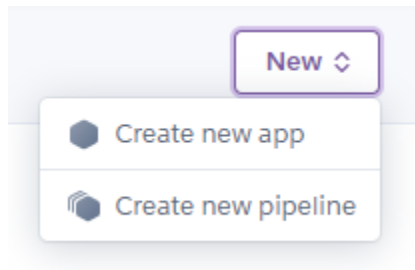
Sau khi cài đặt thành công chúng ta có thể kiểm tra bằng cách:

```
heroku help
```

3.3.5. Tạo và deploy website

Trước khi tiến hành deploy ta tiến hành tạo một app trên Heroku:

Tại <https://dashboard.heroku.com/apps> ta chọn vào mục new app:



Tiến hành tạo app với tên hợp lệ.

Sau khi app được tạo thành công ta chọn vào tab **Deploy** ta có giao diện sau:

Install the Heroku CLI

Download and install the [Heroku CLI](#).

If you haven't already, log in to your Heroku account and follow the prompts to create a new SSH public key.

```
$ heroku login
```

Clone the repository

Use Git to clone k07q's source code to your local machine.

```
$ heroku git:clone -a k07q  
$ cd k07q
```

Deploy your changes

Make some changes to the code you just cloned and deploy them to Heroku using Git.

```
$ git add .  
$ git commit -am "make it better"  
$ git push heroku master
```

Trước khi thực hiện các bước theo hướng dẫn deploy của heroku ta thực hiện các bước sau để tránh gặp lỗi trong quá trình deploy:

Chúng ta chạy các dòng lệnh sau trong command line:

```
heroku config:set DISABLE_COLLECTSTATIC=1
```

```
heroku ps:scale web=1
```

*Tại settings.py thêm vào **ALLOWED_HOSTS** host của app và localhost:*

```
ALLOWED_HOSTS = [ '<app_name>.herokuapp.com', '127.0.0.1' ]
```

Tiếp tục tại settings.py đặt DEBUG = False:

```
DEBUG = True
```

Bước cuối cùng tiến hành các bước trong hướng dẫn tại mục Deploy trên Heroku và tiến hành deploy app.

TÀI LIỆU THAM KHẢO

- [1] D. S. Foundation, "Django documentation," Django Software Foundation, [Online]. Available: <https://docs.djangoproject.com/>.
- [2] P. S. Foundation, "Python documentation," [Online]. Available: <https://docs.python.org/>.
- [3] T. P. G. D. Group, "PostgreSQL documentation," [Online]. Available: <https://www.postgresql.org/docs/>.
- [4] R. Data, "W3Schools Online Web Tutorials," [Online]. Available: <https://www.w3schools.com/>.
- [5] Salesforce.com, "Heroku documentation," [Online]. Available: <https://devcenter.heroku.com/categories/reference>.