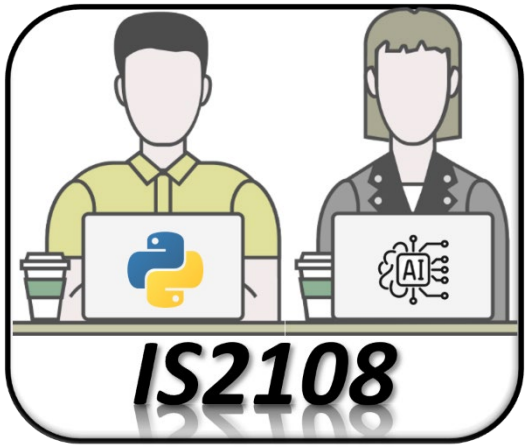


Lecture 5

Web Application Development with Django (I)

IS2108 – Full-stack Software Engineering for AI Solutions I
AY 2025/26 Semester 1

Lecturer: A/P TAN Wee Kek

Email: distwk@nus.edu.sg :: **Tel**: 6516 6731 :: **Office**: COM3-02-35

Consultation: Tuesday, 12 pm to 2 pm. Additional consultations by appointment are welcome.

Learning Objectives

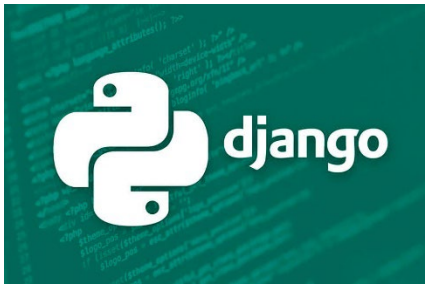
- At the end of this lecture, you should understand:
  - Overview of the Django framework.
  - Architectural pattern of a Django web application – MVT vs. MVC
  - Anatomy of a Django web application.
  - Handling HTTP requests.
  - Working with Django templates.
  - Working with forms.

Readings

- Required readings:
  - None.
- Suggested readings:
  - None

Overview of the Django Framework

- Django is a full-stack framework for Python to develop web applications.
- Django is based on integrated backend and frontend development using server-side rendering.
- For JavaScript development, Next.js is also a full-stack web application framework using server-side rendering:
  - In comparison, Express/React (e.g., MERN) is not full-stack.
  - Express covers backend development.
  - React covers frontend development.
  - Also, Express/React uses client-side rendering.



Overview of the Django Framework (cont.)

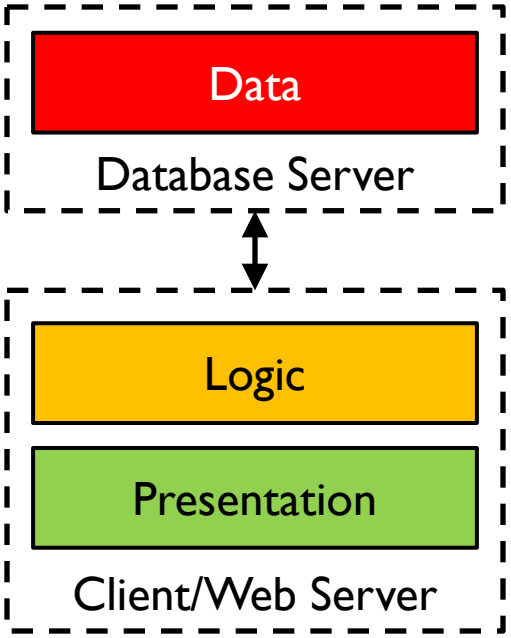
- Pedagogically, Django is a sound framework to start learning web application development with Python:
  - Comprehensive framework that include just about everything that you need.
  - Sound architectural pattern based on Model–View–Template (MVT).
  - Integrated Object-Relational Mapping (ORM).
  - Suitable for building large, scalable, production-grade web applications in enterprise context.

Characteristics of the Django Framework

- Django is a high-level web application framework:
  - It is easy and fast to perform many web application development tasks with Django.
  - E.g., routing, templating, HTTP conversational state management and data persistence.
  - Django also provides many extra features such as security authentication, content administration and caching.
  - Django is highly scalable for supporting large web application with high traffic demand.

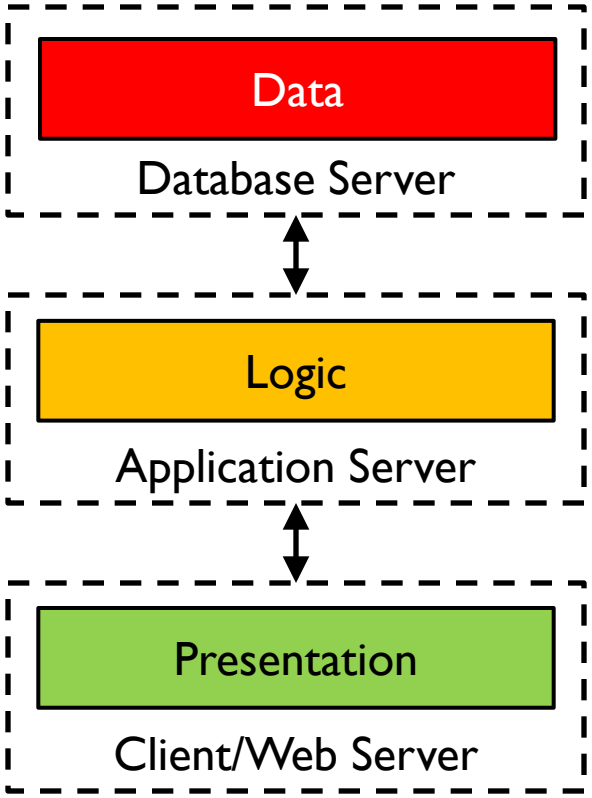
Two-tier Architecture Web Application

- A web application, regardless of whether it uses server-side or client-side rendering, is essentially a two-tier fat-client architecture.
- Fat client:
  - Client handles presentation and logic.
  - Server handles data, mainly refers to database server.
- In server-side rendering, the logic and presentation tiers run on a web server.
- In client-side rendering, the architecture is more complex – *To be discussed in IS3108.*



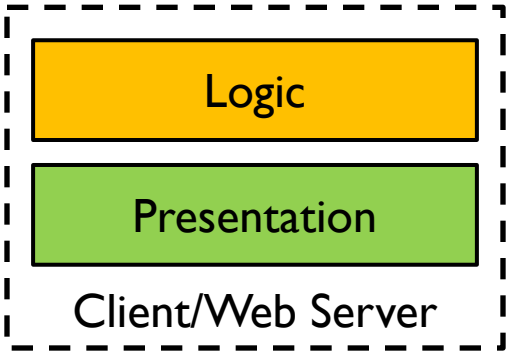
N-tier Architecture Web Application

- In a three-tier architecture:
  - Presentation, logic and data tiers manifest as three separate layers of software applications.
  - There is a distinct separation of logic tier.
- In general, the presentation tier goes through a common backend to manipulate data or process transactions.
- Since Django is full-stack, it typically uses two-tier architecture.
- But Django can support multitier architecture – *To be discussed in future lectures.*



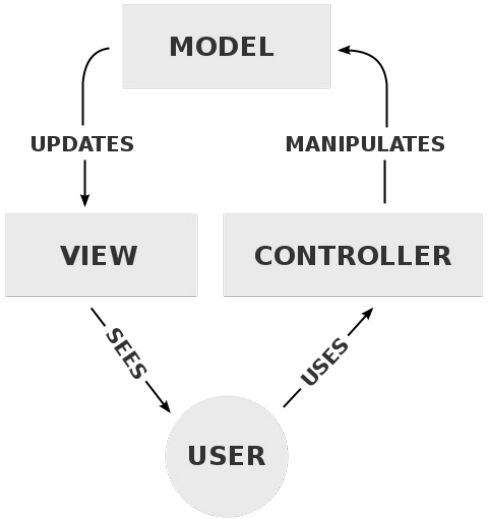
Back to Two-tier Architecture Web Application

- For now, we will focus on two-tier web application…
- Recall that that in multitier architecture, the logic tier is constructed using software components or services.
- Within a two-tier web application:
  - An appropriate architecture must also be used for the presentation tier code to interact with the logic tier code.
  - We will defer discussion on the interaction between the logic tier and data tier to a future lecture.



Model-View-Controller (MVC)

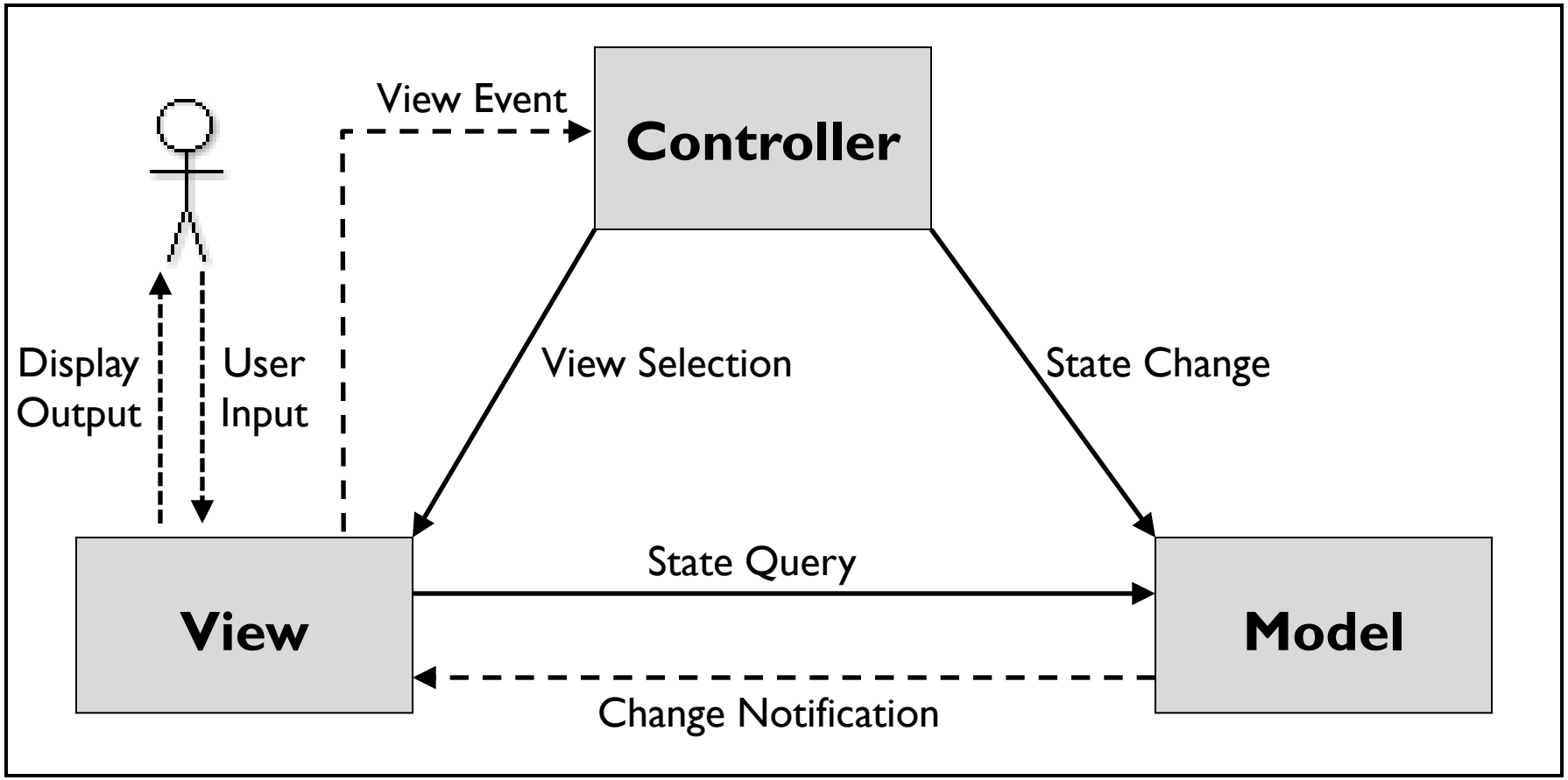
- **Model-View-Controller (MVC)** is a software architectural pattern for front-end applications:
  - Used for developing user interfaces that divides an application into three interconnected parts.
  - Separate internal representations of information from the ways information is presented to, and accepted from, the user.
  - Decoupling of components allow for efficient code reuse and parallel development.
- There are various variations of the MVC architecture.



Model-View-Controller (MVC) (cont.)

- **Model**:
  - Manages the behaviour and data of the application domain.
  - Responds to requests for information about its state (usually from the view).
  - Responds to instructions to change state (usually from the controller).
- **View**:
  - Manages the display of information.
- **Controller**:
  - Interprets the mouse and keyboard inputs from the user.
  - Inform the model and/or the view to change as appropriate.

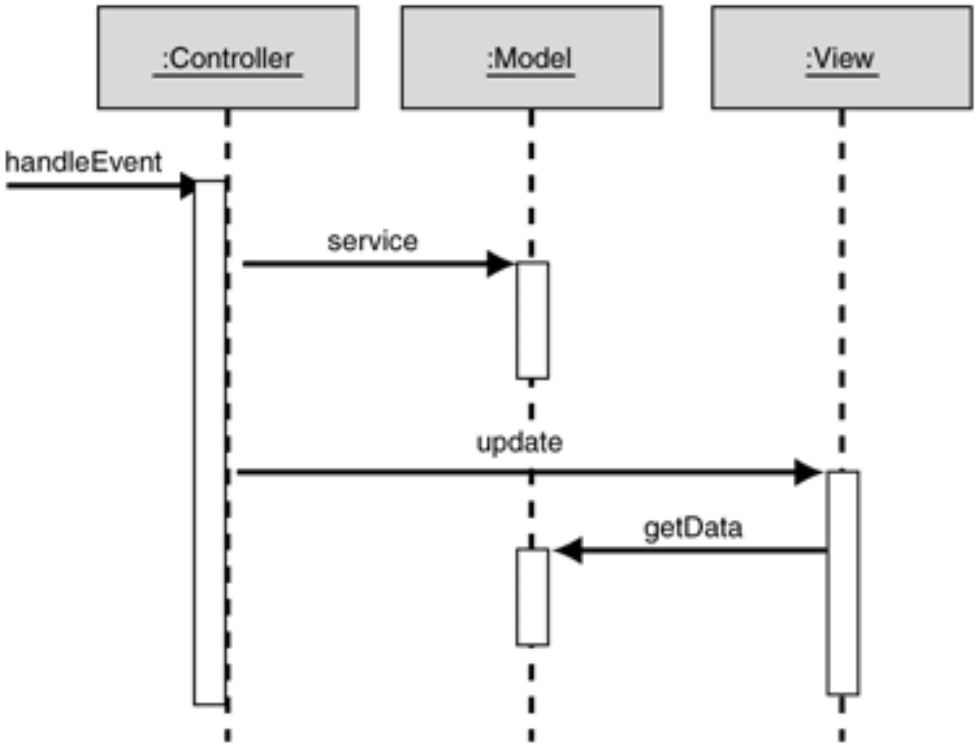
Model-View-Controller (MVC) (cont.)



Method Invocation (direct association)

Events (indirect association)

Model-View-Controller (MVC) (cont.)



Model-View-Template (MVT)

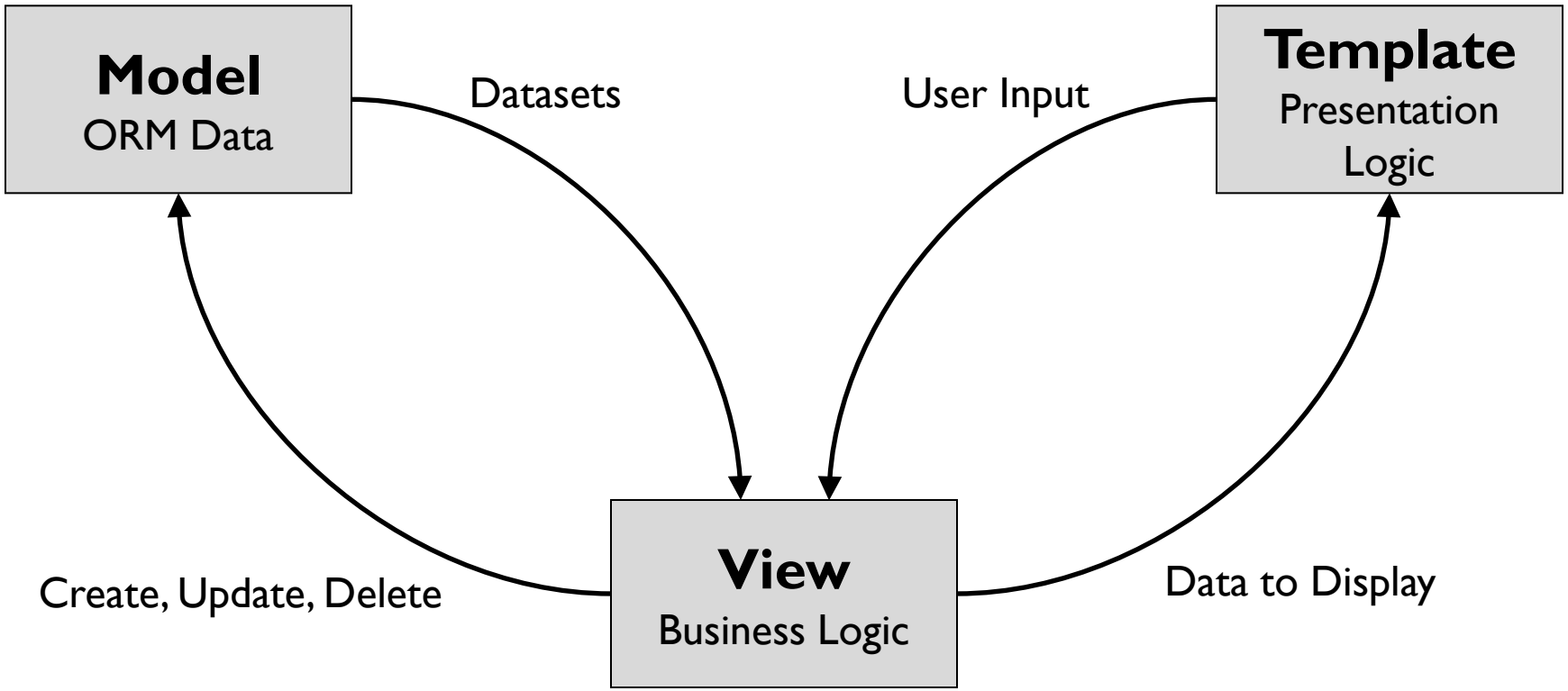
- Django is based on the MVT architectural pattern, a variation of MVC.
- **Model**:
  - Represents the data layer.
  - Defines the data schema, relationships, and database interactions.
  - Implemented using Django's ORM.
  - Example – A `Student` model defined in `models.py`.
- **View**:
  - Handles the business logic of the application.
  - Receives HTTP requests, interacts with the model if needed, and passes context to the template.

Model-View-Template (MVT) (cont.)

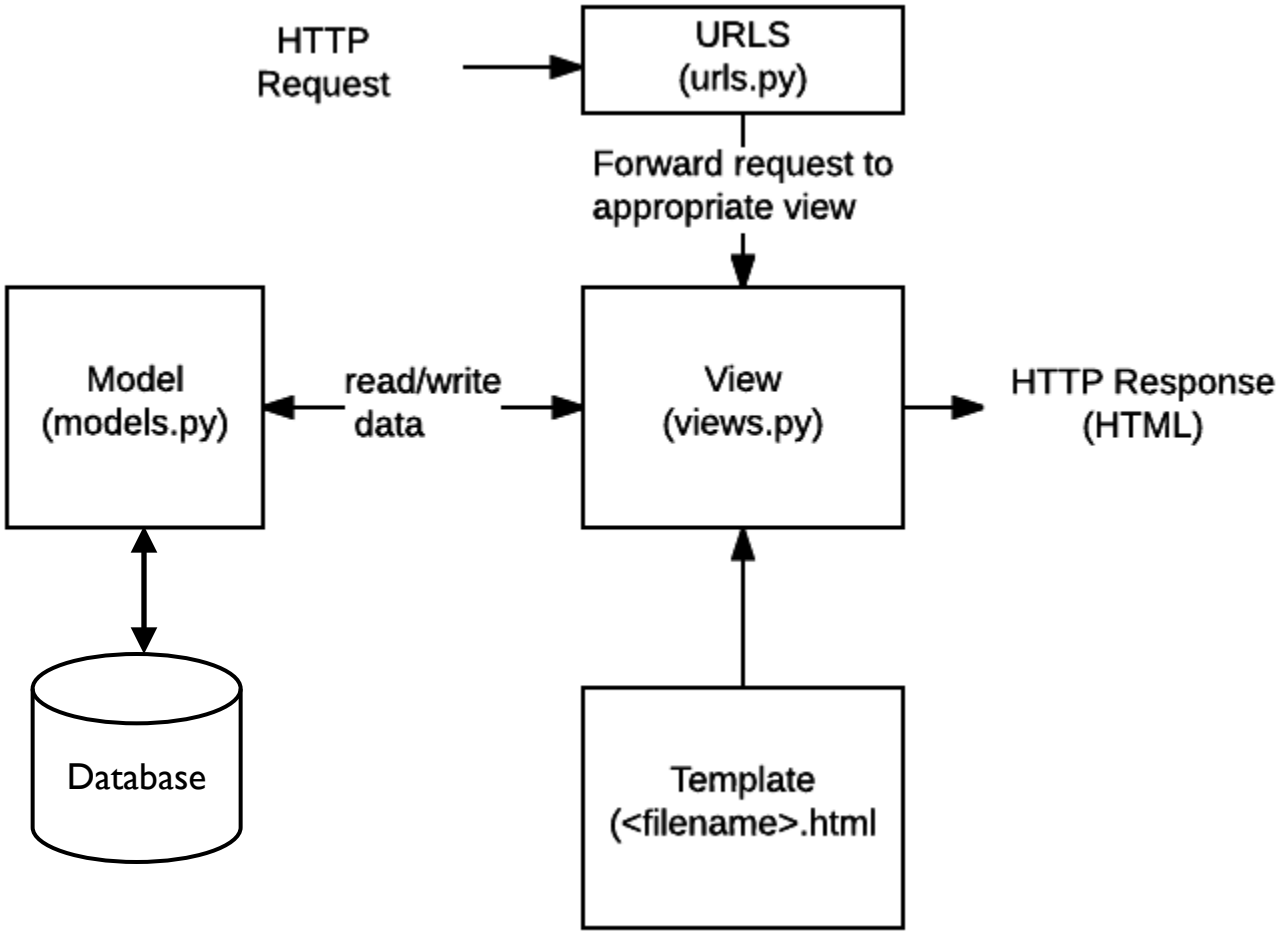
- Unlike MVC, Django views don't directly handle presentation.
- Views in Django act more like controllers in traditional MVC.
- Example – A function or class-based view in `views.py` for manipulating the `Student` model.

- **Template**:
  - Defines the actual presentation layer.
  - Uses Django's template language to generate dynamic HTML for the user.
  - Example: `student.html` displaying student data passed from the view.

Model-View-Template (MVT) (cont.)



Model-View-Template (MVT) (cont.)



Model-View-Template (MVT) (cont.)

- In Django, the use of models are not essential:
  - You only need to define Django models if you are storing, retrieving, or manipulating data in a database.
  - Note that Django's ORM is tightly integrated into the framework:
    - Forms, admin, authentication, and many third-party apps assume the presence of models.
    - So, there are advantages with using Django models.
  - You do not need to use Django models for some scenarios:
    - Static content websites.
    - Computationally heavy application – In-memory data.

Installing Django

- An official release of Django can be installed with `pip`:

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

  - https://www.djangoproject.com/
  - The latest long-term support (LTS) version is 5.2.x
  - Works with Python 3.10 or later.
- To verify the successful installation of Django, running the following code in a `.py` file:

```
import django
print(django.get_version())
```

Installing Django (cont.)

- Django uses SQLite by default, and you only need to install and set up a server-based database, if required:
  - E.g., MySQL or PostgreSQL.
  - Refer to Django's website for more information.

Django Terminologies

- In Django, a web application is known as a project:
  - A Django project is a collection of settings, configurations, and apps that together make up a full web application.
  - It also defines the global environment in which one or more Django apps run.
- Within a project, you can create apps:
  - A project is the whole website or system (e.g., an e-commerce site).
  - An app is a component of that project designed for a specific purpose (e.g., a shopping cart app, a payments app, a user authentication app).

Django Terminologies (cont.)

- Example of a project for a library management system:
  - Project – library_system (contains global settings and configuration).
  - Apps inside project:
    - catalogue – handles books and categories.
    - members – manages library users.
    - loans – manages book borrowing and returns.
- A single project can have many apps:
  - Apps can be reused across multiple projects.
- In summary:
  - Django project is the overarching configuration and container for your entire web application.
  - Apps are modular building blocks inside the project.

Django Project with Python Virtual Environment

- It is recommended to use Python virtual environment to manage the set of packages that a project requires:
  - There are two options to set up virtual environment.
- venv outside the project (recommended):
  - Cleaner project folder.
  - Easy to recreate venv with pip:

```
pip freeze > requirements.txt
pip install -r requirements.txt
```

  - Works well with Git without the need to ignore the whole venv.
  - Better for code distribution.

Django Project with Python Virtual Environment (cont.)

```
projects/
│
├── mysite/              ← Django project (code, apps, settings, etc.)
│   ├── manage.py
│   ├── mysite/
│   └── apps/
│
└── venv/                ← Virtual environment (not in repo)
```

- venv inside the project:
  - More convenient for self-contained project folder.
  - If using Git, must ignore venv/

```
mysite/
├── venv/                ← Virtual environment (inside project)
├── manage.py
└── mysite/
```

Creating a New Django Project

- Initial setting up:
  - Create parent folder.

```
mkdir lecture05
cd lecture05
```

  - Create venv

```
python -m venv venv
```

  - Activate venv

```
.\venv\Scripts\activate.bat
```

    - The above commands are based on Windows.
    - To activate venv in macOS:

```
source ./venv/bin/activate
```

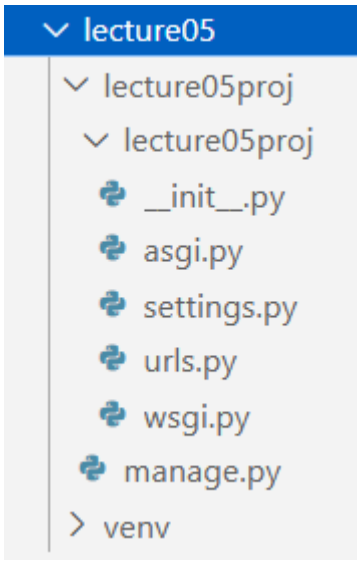
  - Install Django with `pip` within the venv.

Creating a New Django Project (cont.)

- Creating a new Django project:
  - Use `django-admin` to bootstrap a new project:

```
django-admin startproject lecture05proj
```

  - The above command will create a project called `lecture05proj` inside the `lecture05` parent folder.



Anatomy of a Django Project

- The `django-admin startproject` command creates various files.
- `manage.py`
  - A command-line utility for interacting with the Django project.
- `lecture05proj/`
  - A folder containing the actual Python package for the project.
  - The folder name is the Python package name for importing anything inside it (e.g., mysite.urls)
- `lecture05proj/__init__.py`
  - An empty file that tells Python that this directory should be considered a Python package.

Anatomy of a Django Project (cont.)

- `lecture05proj/settings.py`
  - Settings/configuration for this Django project.
- `lecture05proj/urls.py`
  - The URL declarations for this Django project.
  - It resembles a "table of contents" of the website.
- `lecture05proj/asgi.py`
  - An entry-point for ASGI-compatible web servers to serve this project.
- `lecture05proj/wsgi.py`
  - An entry-point for WSGI-compatible web servers to serve this project.

Running a Django Project

- Django development server:
  - A built-in lightweight web server for running and testing Django projects locally.
  - Does not require the installation of Apache, Nginx or Gunicorn.
  - Key features:
    - The server restarts automatically when you save changes to your Python code.
    - Supports debug mode.
    - Supports static file serving.

Running a Django Project (cont.)

- To start the development server:
  - Change to the `lecture05proj/` folder.
  - Run the following command:

```
python manage.py runserver
```

  - By default, the development server will be started on port 8000.
  - You can view the project's default web page at http://localhost:8000/

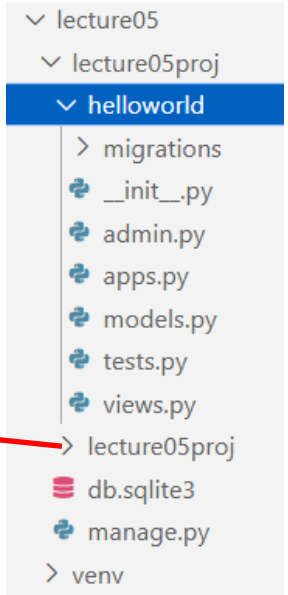
Create a Django App in a Project

- To create a new `helloworld` app in `lecture05proj`:
  - Change to the `lecture05proj/` folder.
  - Run the following command:

```
python manage.py startapp helloworld
```

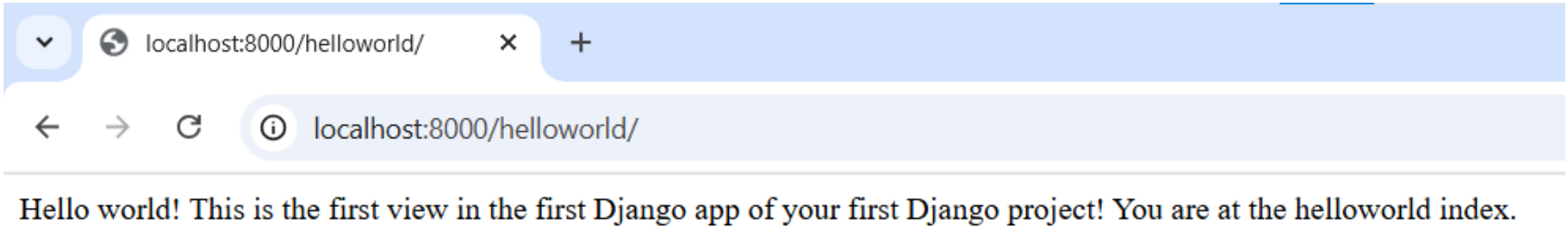
  - The above command will create a new `helloworld` folder structure with the various files.
  - The `helloworld` folder essentially contains the app.
- Edit `lecture05proj/lecture05proj/settings.py` to add the new app under `INSTALLED_APPS`:

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



Create a Django App in a Project (cont.)

- Creating a new view for the `helloworld` app involves the following steps:
  - Edit `helloworld/views.py` to define a Python function for the view.
  - Edit `helloworld/urls.py` to define a `URLconf` for the `helloworld` app.
  - Edit `lecture05proj/urls.py` to configure the root `URLconf` to include the one for the `helloworld` app.



Handling HTTP Requests

- A web application handles requests for HTML content at a route, path or URL (uniform resource locator):
  - In Django, the term URL is more commonly used.
  - URLs are defined in a Django app using a Python module informally called a `URLconf` (URL configuration).
  - This module contains pure Python code and is a mapping between URL path expressions and Python functions (the views).
- URLs in Django are defined in a hierarchical manner:
  - URLs to apps are defined in the project-level `urls.py`.
  - URLs within an app are defined in the app-level `urls.py`.

Handling HTTP Requests (cont.)

- When a user requests a page from a Django site, Django processes the request using the following algorithm:
  1. Determines the root `URLconf` module to use.
  2. Django loads that Python module and looks for the variable `urlpatterns`:
     - This variables should be a sequence of `django.urls.path()` and/or `django.urls.re_path()` instances.
  3. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL.
  4. Once one of the URL patterns matches, Django imports and calls the given view:
     - The view is a Python function or a class-based view.
     - The view is passed an instance of `HttpRequest`.

Handling HTTP Requests (cont.)

- The keyword arguments are made up of any named parts matched by the path expression.
- If there is no named groups, then the matches from the regular expression are provided as positional arguments.

5. If no URL pattern matches, Django invokes an appropriate error-handling view:
   - An exception raised during any point in this process will also trigger error-handling view.

- Important points to note:
  - Use angle brackets `<>` to capture a value from the URL.
  - Captured values can optionally include a converter type:
    - E.g., use `<int:name>` to capture an integer parameter.
  - No need to add a leading slash to a URL.

Handling HTTP Requests (cont.)

- Query string parameters:
  - Django's URL dispatcher `path()` only matches the path part of URL.
  - It does not handle query string parameters.
  - Query string parameters are available inside the view via the `request.GET` dictionary.
- For search engine optimisation (SEO), the use of query string parameters should be avoided.

Handling HTTP Requests (cont.)

- Let's examine some concrete examples:

```
7  urlpatterns = [
8      path('', views.index, name='index'),
9      path('demo01/<name>/<age>', views.demo01Echo),
10     path('demo02/<year>/<month>/<day>', views.demo02Date),
11     path('demo03/<int:year>/<int:month>/<int:day>', views.demo03DateWithConverters),
12     path('demo04/', views.demo04QueryStringParams),
13 ]
```

  - `demo01` treats name and age as positional arguments in which the order of the values cannot be changed.
  - `demo02` treats year, month and day as named arguments in which the order of the values can be changed.
  - `demo03` demonstrates the use of converters.
  - `demo04` demonstrates the use of query string parameters.

Handling HTTP Requests (cont.)

```
14  def demo01Echo(request, name, age):
15
16      return HttpResponse(f'demo01Echo: Named arguments received: {name} is {age} years old.')
17
18
19
20  def demo02Date(request, day, month, year):
21
22      print(type(year), type(month), type(day))
23
24      return HttpResponse(f'demo02Date: Named arguments received: {year}-{month}-{day}')
25
26
27
28  def demo03DateWithConverters(request, day, month, year):
29
30      print(type(year), type(month), type(day))
31
32      return HttpResponse(f'demo03DateWithConverters: Named arguments received: {year}-{month}-{day}')
33
34
35
36  def demo04QueryStringParams(request):
37
38      # demos/demo04/?name=Bob&age=12
39
40      name = request.GET.get('name', 'unknown')
41      age = request.GET.get('age', 'unknown')
42
43      return HttpResponse(f'demo04QueryStringParams: Query string parameters received: {name} is {age} years old.')
```

Using Regular Expressions in Django URLs

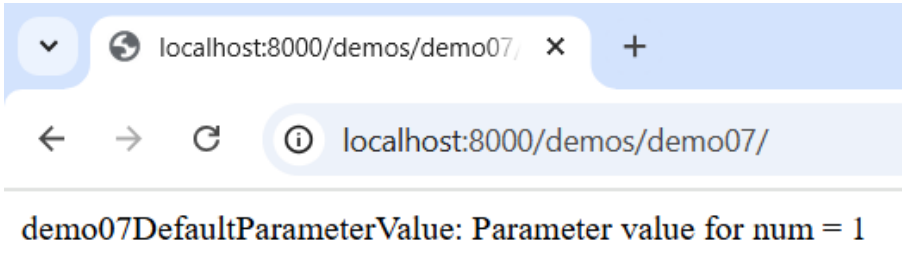
- Regular expressions can be used to define URLs using the request dispatcher `re_path()` instead of `path()`.
- Named regular expression groups:
  - The syntax is `(?P<id>pattern)`
  - `name` is the name of the group
  - `pattern` is some pattern to match.
  - E.g., `(?P<id>\d+)` will match one or more digits that represents a numerical identifier.
  - See `demo05` for the example.
- Unnamed regular expression groups can be specified without the angle brackets `<>`:
  - See `demo06` for the example.

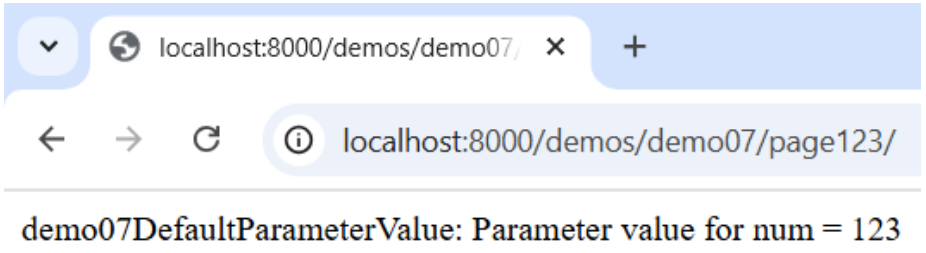
Specifying Defaults for View Arguments

- Default parameter value for a view's argument:
  - Can be specified using the standard Python optional argument syntax.
  - In Python, no default value implies mandatory argument.
- See `demo07` for an example.

```
15      path('demo07/', views.demo07DefaultParameterValue),
16      path('demo07/page<int:num>/', views.demo07DefaultParameterValue),

59  def demo07DefaultParameterValue(request, num=1):
60
61      return HttpResponse(f'demo07DefaultParameterValue: Parameter value for num = {num}')
```





Django Templates

- Observed that using a Python function to return the HTML view has several disadvantages:
  - Difficult and low readability when generating long HTML content.
  - Difficult to interspersed dynamic data.
  - Difficult to generate HTML content conditionally and iteratively.
- Django provides a convenient way to generate HTML dynamically using templates:
  - A Django project can be configured to use zero, one or more template engines.

Django Templates (cont.)

- The built-in template engine is Django Template Language (DTL).
- Other popular alternative template engines such as Jinja2 are also supported.

- Django defines a standard API for loading and rendering templates regardless of the backend:
  - Loading consists of finding the template for a given identifier and preprocessing it (compile to an in-memory representation).
  - Rendering means interpolating the template with context data and returning the resulting string.

Django Templates (cont.)

- Where should templates be placed?
  - App-specific templates can be placed in a `templates/app_name` folder of the respective app:
    - In `project_name/project_name/settings.py`, ensure that `TEMPLATES.APP_DIRS` is set to `True`.
  - Project-wide or shared templates can be placed in a `templates/` folder of the project.

Django Template Language (DTL)

- Django template:
  - A text document or a Python string marked-up using the Django template language.
  - Text document is more useful.
- A template is rendered with a context:
  - Rendering replaces variables with their values, which are looked up in the context.
  - The tags are then executed.
  - Everything else is output as is, e.g., static HTML.
- The syntax of the Django template language involves four constructs.

Django Template Language (DTL) (cont.)

- **Variables**:
  - A variable outputs a value from the context.
  - Context consists of a dict-like object that maps keys to values.
  - Variables are surrounded by `{{` and `}}`
  - Example:
    - The following temple:

My first name is `{{ first_name }}`. My last name is `{{ last_name }}`.

    - With the context:

```
{'first_name': 'John', 'last_name': 'Doe'}
```

    - Will be rendered into:

My first name is John. My last name is Doe.

Django Template Language (DTL) (cont.)

- See `demo08` for an example of rendering variables using context:
  - In this demonstration, the context is populated using parameters from the HTTP request.

```python
def demo08DjangoTemplateLanguageVariables(request, name, age):

    context = {
        'name': name,
        'age': age,
    }

    return render(request, 'demos/demo08.html', context)
```

```html
<html>
    <head>
        <title>Demo 08 - Django Template Language - Variables</title>
    </head>
    <body>
        <h1>Demo 08 - Django Template Language - Variables</h1>
        <p>Hello, {{ name }}!</p>
        <p>Your age is {{ age }}.</p>
    </body>
</html>
```

Django Template Language (DTL) (cont.)

- **Tags**:
  - Tags provide arbitrary logic in the rendering process.
  - By definition, tags are versatile and can be used for various purposes such as conditional control flow, iterative control flow or database retrieval.
  - Variables are surrounded by `{%` and `%}`
  - Tags can accept arguments after the tag name, e.g., `{% tagname 'arg1' 'arg2' %}`
  - Some tags require beginning and ending tags, e.g., the `if` tag is written as `{% if %} … {% endif %}`

Django Template Language (DTL) (cont.)

- **Filters**:
  - Filters are used to transform the values of variables and tag arguments.
  - E.g., `{{ foo|title }}` transforms the string value of the variable `foo` into title casing.
  - Filters can accept argument.
  - E.g., `{{ my_date|date:"Y-m-d" }}` accept the format for the `date` filter.
  - Django provides built-in filters, and it is also possible to create custom filters.
- **Comments**:
  - Comments are surrounded by `{#` and `#}`

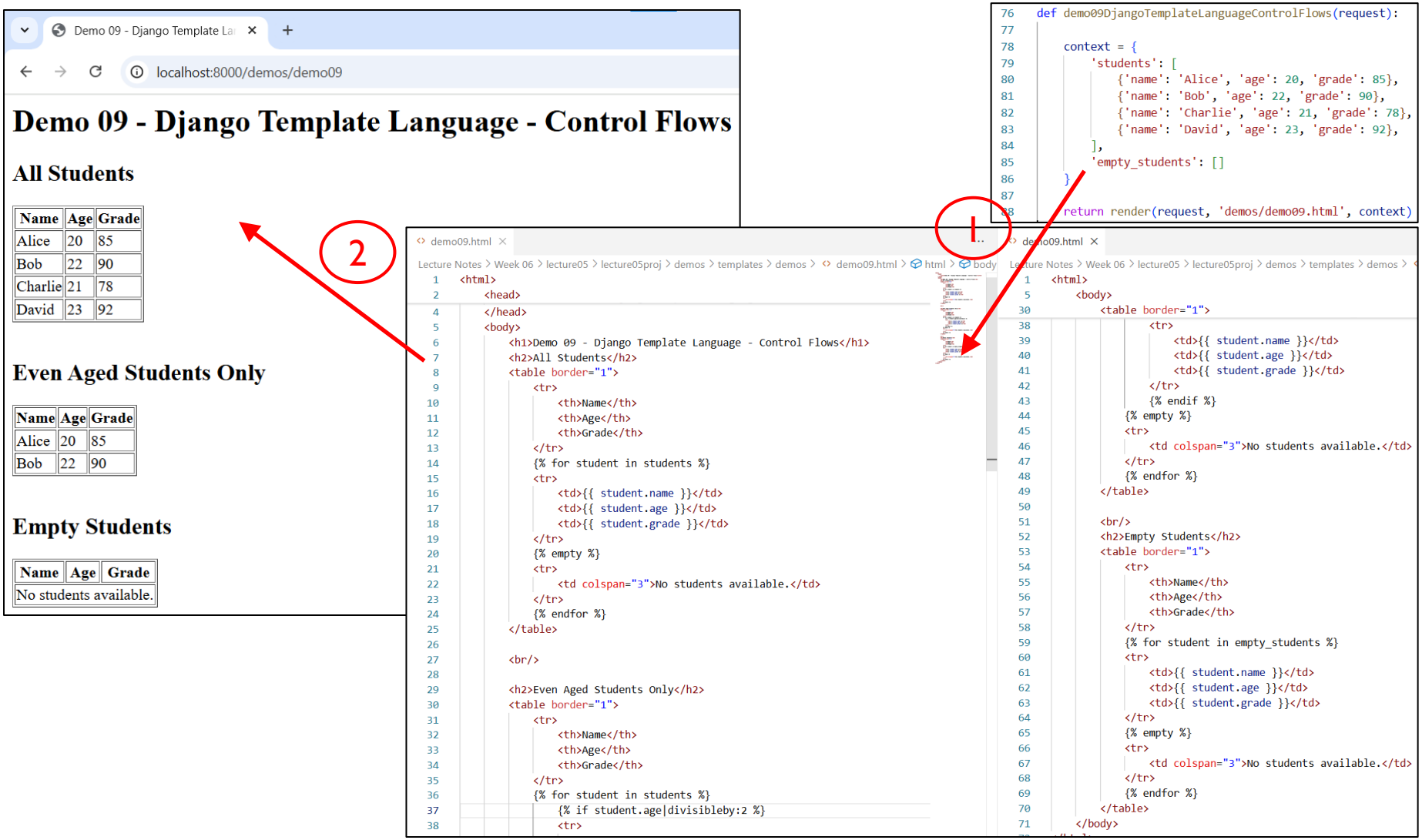
Using Control Flows in DTL

- **Conditional branching**:
  - Standard conditional block is written using `{% if %} … {% endif %}`
  - Branching is written using `{% if … elif … else %}`
- **Iteration or loops**:
  - Iterating over a sequence is written using `{% for … in … %} … {% endfor %}`
  - If the loop is empty, a fallback can be specified using `{% empty %}` inside a `{% for %}` loop.
  - Loop control can be perform using `{% break %}` and `{% continue %}`

Using Control Flows in DTL (cont.)

- Observe that the syntaxes of the control flows resemble those of the Python language construct.
- Django does not support the while loop:
  - DTL is designed to be presentation-focused.
  - Avoid infinite loop.
  - Complex business logic should be placed in the view and not the template.
- See `demo09` for examples of rendering dynamic content using DTL's control flows.

Using Control Flows in DTL (cont.)



Working with Forms in Django

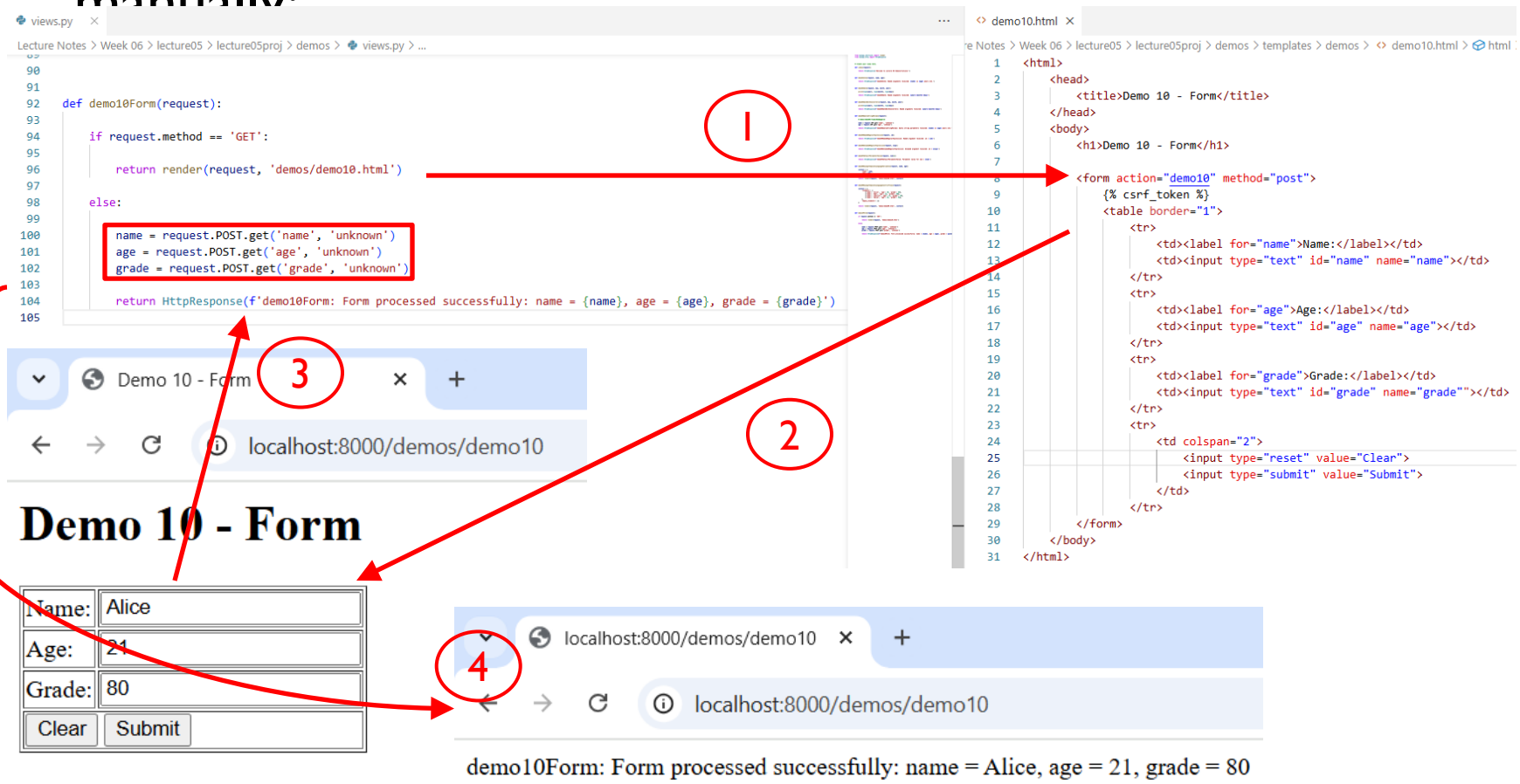
- Most web applications will need to accept input data from users to facilitate processing:
  - Login to a website.
  - Manage shopping cart.
  - Perform CRUD data operations.
- This will require the use of forms:
  - Recall that a HTML form consists of a collection of input elements for users to provide input data.
  - The input data in a HTML form needs to be sent to the server for processing using HTTP GET and POST.
  - Handling forms involves complex logic, but Django provides great support for form processing.

Working with Forms in Django (cont.)

- Django handles three distinct parts of form processing:
  - Preparing and restructuring data to make it ready for rendering.
  - Creating HTML forms for the data.
  - Receiving and processing submitted forms and data from the client.
- Django provides various approaches to process forms with and without the use of model:
  - These approaches will be discussed in a future lecture.
- The most basic approach is to parse the form data manually using `request.POST`.

Working with Forms in Django (cont.)

- See `demo10` for an example of parsing form input data manually:



Summary

- Django is a full-stack web application development framework for Python using server-side rendering.
- Django is a high-level framework that provides excellent support for various web application development tasks.
- These include HTTP request handling, templating and form processing.

Q&A

Next Lecture…

- Learn about:
  - Overview of object-oriented programming with Python.
  - Advanced concepts on web application development with Django.

In-class Formative Quiz for Lecture 05