Chapter 1.1 – Introduction to Python

Python Basics

- Topics
  - Introduction
  - Data Types
  - Arithmetic/Logic Operations

Notable Features

- Many basic data types : numbers (floating point, complex, and unlimited-length long integers), strings, lists, and dictionaries.
- Supports object-oriented programming: with classes and multiple inheritance.
- Supports raising and catching exceptions: cleaner error handling.
- Strongly and dynamically typed data types: mixing incompatible types (e.g. attempting to add a string and a number) causes an exception, errors caught sooner.
- Automatic memory management: frees you from having to manually allocate and free memory in your code.

Notable Feautures

- Elegant syntax: programs easier to read.
- Easy-to-use language: ideal for prototype development.
- Large standard library: supports many common programming tasks e.g. connecting to web servers, regular expressions, file I/O.
- A bundled development environment called IDLE.
- Runs on different computers and operating systems: Windows, MacOS, many brands of Unix, OS/2, ...
- Free software: Free to download or use Python - the language is copyrighted it's available under an open source license.

Indentation

- Python does not use brackets to structure code, instead it uses whitespaces
  - Tabs are not permitted.
    - Four spaces are required to create a new block,
  - To end a block simply move the cursor four positions left.
  - An example:

```
1. def bar(x):
2.     if x == 0:
3.         foo()
4.     else:
5.         foobar(x)
```


Indentation

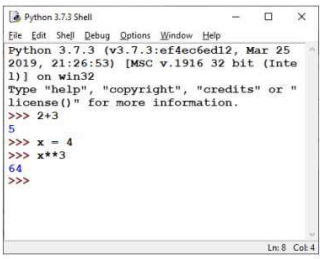
- Python does not use brackets to structure code, instead it uses whitespaces
  - Tabs are not permitted.
    - Four spaces are required to create a new block,
  - To end a block simply move the cursor four positions left.
  - An example:

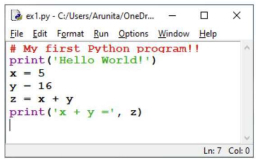
```
1. def bar(x):
2.     if x == 0:
3.         foo()
4.     else:
5.         foobar(x)
```

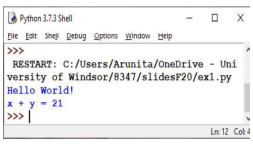

IDLE EXAMPLE

IDLE

- IDLE: Basic IDE that comes with Python
  - Should be available from Start Menu under Python program group.
  - Main "Interpreter" window.
    - Allows us to enter commands directly into Python
    - As soon as we enter in a command Python will execute it display the result.
    - '>>>' signs act as a prompt.







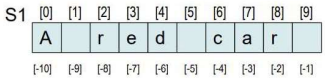
Numeric Data Types

- *int*: represents positive and negative whole numbers.
  - Written without a decimal point
  - e.g. 5, 258964785663
- *float*: written with a decimal point
  - e.g. 3.0, 5.8421, 0.0, -32.5 etc

Arithmetic Operators

- Basic arithmetic operators: + (addition), - (subtraction), * (multiplication), / (division)
  - / (division) produces floating point value 15/3 → 5.0
  - // (integer division) truncates any fractional part 25//3 → 8
  - % (remainder) gives the remainder after integer division. 25%3 → 1
  - Augmented assignment operators: +=, -=, *=, /=

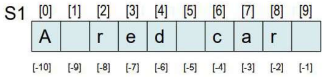
String Methods



- S1.count('r')
- **2**
- S1.split()
- **['A', 'red', 'car']**
- S1.replace('r', '*')
- **"A *ed ca* "**
- S1.upper()
- **"A RED CAR "**

NOTE: This is not an exhaustive list

String Methods



- S1.index('r')
- 2
- S1.index('x')
- ValueError
- S1.find('r')
- 2
- S1.find('x')
- -1
- S1.startswith('A red')
- True

NOTE: This is not an exhaustive list

Lists

- A collection data type that is ordered and changeable (mutable).
  - e.g. [1], [3, 'hi', 4.5, [1,2,3]], []
- Use [] to index items; similar to strings
- Can have multiple indices, e.g. list1[2][0], list1[-2][-1][-3]

Lists (Examples)

```
Python 3.8.3 Shell
File Edit Shell Debug Options Window Help
>>> list1 = [1, 2.5, 'hello class', [2, 18, 12, 'house'], -8]
>>> list1[1]
2.5
>>> list1[2][0]
'h'
>>> list1[-2][-1][0:3]
'hou'
>>> list1[15]
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    list1[15]
IndexError: list index out of range
>>> list1[2:15]
['hello class', [2, 18, 12, 'house'], -8]
>>>
```

Your Turn …

- L1 = [1], [3, 'hi', 4.5, [1,2,3]]
  - >>> L1[1]
  - >>> len(L1)
  - >>> L1[3][0]
  - >>> L1[6]
  - >>> L1[-1]

range(n)

- *range(n)*: produces sequence 0, 1, 2, … *n*-1
- range([*i*,]*stop*[, *k*]): sequence starts at *i* (instead of 0) and incremented by *k* (instead of 1)
- range(5) → produces sequence 0, 1, 2, 3, 4
- range(3, 20, 4) → produces 3, 7, 11, 15, 19
- >>> L = [1, 5, 7, 2, 8]
- >>> for i in range(len(L)):
  L[i] = L[i]+10
- >>> L
  - [11, 15, 17, 12, 18]

List Methods

- L = [1,2,3,[4,5,'a','b'], 'hello', '6', 7]
  - L.append([8,9])
  - **[[1, 2, 3, [4, 5, 'a', 'b'], 'hello', '6', 7, [8, 9]]**
  - L += [8,9]
  - **[1, 2, 3, [4, 5, 'a', 'b'], 'hello', '6', 7, 8, 9]**
  - L.index(3)
  - **2**

NOTE: This is not an exhaustive list. We are always starting with initial value of L.

Dictionaries

- *dict*: an ***unordered*** collection of key-value pairs
  - ***mutable***
  - unordered → no notion of index positions
  - keys are unique → a key-value item ***replaces*** existing item with same key
  - duplicate values are fine

- d1 = {"name": "John", "age": 25, (1,2):['a', 'b', 'c'], 0: "blue", 86: 20, 1:20}
- d2 = dict([("name","John"), ("age", 25), ((1,2), ['a', 'b', 'c']), (0, "blue"), (86, 20)])

Chapter 1.3 – Python Programming(Part – 3)

Python Basics

- Topics
  - If statements
  - Loops
  - Exception handling
  - Functions
  - File Input/Output
  - Modules

Control Flow

- Conditional branching
  - IF statement
- Looping
  - While
  - For … in
- Exception handling
- Function or method call

IF Statement

- *suite*: a block of code, i.e. a sequence of one or more statements
- Syntax:

```
if bool_expression1:
   suite1
elif bool_expression2:
    suite2
…
elif bool_expressionN:
    suiteN
else:
    else_suite
```

- No parenthesis or braces
- **:** used whenever a suite is to follow
- Use *indentation* for block structure

```
if a < 10:
    print("few")
elif a < 25:
    print("some")
else:
    print("many")
```

While Statement

- Used to execute a suite 0 or more times
  - number of times depends on while loop's Boolean expression.
  - Syntax:

```
while bool_expression:
        suite
```

  - Example:

```
x, sum = 0,0
while x < 10:
        sum += x
        x += 2
print(sum, x)   #What is final value of sum and x
```

Answer: sum=20, x=10

For … in Statement

- Syntax:

```
for variable in iterable:
        suite
```

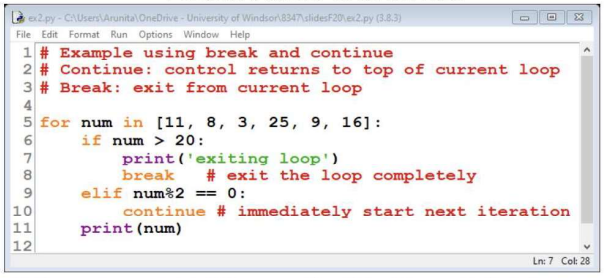
- Example: fruits = ['apple', 'pear', 'plum', 'peach']

```
for item in fruits:
        print(item)
```

- Alternatively

```
for i in range(len(fruits)):
        print(fruits[i])
```

Break and Continue



```
# Example using break and continue
# Continue: control returns to top of current loop
# Break: exit from current loop

for num in [11, 8, 3, 25, 9, 16]:
    if num > 20:
        print('exiting loop')
        break   # exit the loop completely
    elif num%2 == 0:
        continue # immediately start next iteration
    print(num)
```

Iter#	num	num>20?	Num%2==0?	print(num)
0	11	n	n	11
1	8	n	y	
2	3	n	n	3
3	25	y		

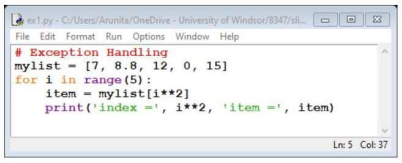
Exception Handling

- Functions or methods indicate errors or other important events by ***raising exceptions***.
- Syntax (simplified):

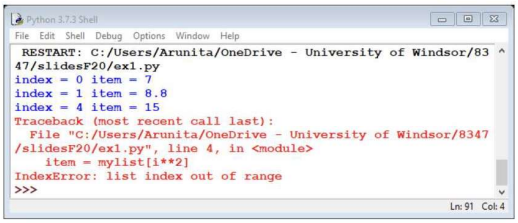
```
try:
   try_suite
except exception1 as variable1:
   exception_suite1
…
except exceptionN as variableN
finally:
   # cleanup
```

- variable part is optional

Exception Handling – Example 1

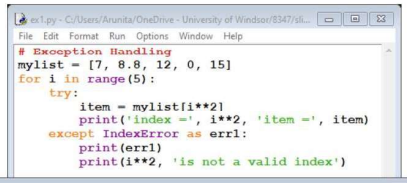


```
# Exception Handling
mylist = [7, 8.8, 12, 0, 15]
for i in range(5):
    item = mylist[i**2]
    print('index =', i**2, 'item =', item)
```

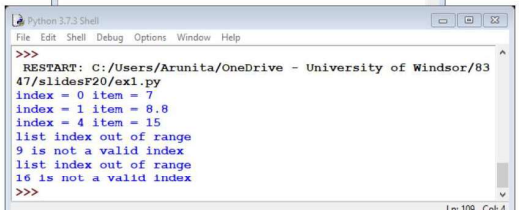


```
RESTART: C:/Users/Arunita/OneDrive - University of Windsor/8347/slidesF20/ex1.py
index = 0 item = 7
index = 1 item = 8.8
index = 4 item = 15
Traceback (most recent call last):
  File "C:/Users/Arunita/OneDrive - University of Windsor/8347/slidesF20/ex1.py", line 4, in <module>
    item = mylist[i**2]
IndexError: list index out of range
>>>
```

Exception Handling – Example 1



```
# Exception Handling
mylist = [7, 8.8, 12, 0, 15]
for i in range(5):
    try:
        item = mylist[i**2]
        print('index =', i**2, 'item =', item)
    except IndexError as err1:
        print(err1)
        print(i**2, 'is not a valid index')
```



```
>>>
RESTART: C:/Users/Arunita/OneDrive - University of Windsor/8347/slidesF20/ex1.py
index = 0 item = 7
index = 1 item = 8.8
index = 4 item = 15
list index out of range
9 is not a valid index
list index out of range
16 is not a valid index
>>>
```

Dynamic Web Sites

- Topics
  - Introduction
  - Communication
    - HTTP, URL
  - Presentation
    - HTML/CSS, templates
  - Overall Structure (MVC)
    - First look at Django

Client Server Model

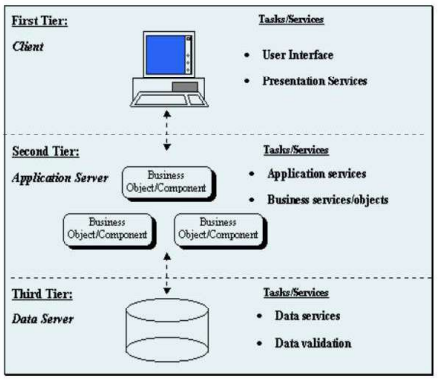


*Fig. taken from [1]

- User requests document from the Web server.
- Web server fetches and/or generates the necessary document.
- The result is returned to the user's browser.
- The browser renders the document.

Static vs. Dynamic Web Pages

- *Static web page*: requests to the same URL always return the same information.
  - Content consists of HTML text files stored on the server.
  - URL typically does not contain parameters; simply a 'path'
  - Primarily informational
- *Dynamic web page*: Data returned by a given URL can vary significantly.
  - generates content and displays it based on actions the users make on the page
  - Functional and informational

HTTP

- *HTTP*: Hyper-Text Transfer Protocol
  - Encapsulates the process of serving web pages
  - Protocol for client-server communication
  - Most clients/servers use version 1.1; HTTP 2 gaining popularity.
  - ***A network protocol*: defines rules and conventions for communication between network devices.
- HTTP is stateless
  - Server maintains no information on past client requests.

URL

- *URL*: Uniform Resource Locator
- General Format: *<scheme> : //<host> :<port> /<path> ;<parameters> ?<query>*
  - Scheme: Protocol being used (e.g. http)
  - Host: host name or IP address
  - Port: TCP port number used by the server (if not specified, defaults to 80 for http)
  - Query passes parameters
  - Example:
    - https://www.google.ca/?gfe_rd=cr&ei=XzYUVceeHayC8QfamoGgDw&gws_rd=ssl#q=http

HTTP

- Application level protocol
  - Client sends request
  - Server responds with reply
  - Other application level protocols are FTP, SMTP, POP etc.
- Almost always run over TCP
  - Uses 'well known' port 80 (443 secure)
  - Other ports can be used as well
  - Can support multiple request-reply exchanges over a single connection

HTTP Message

- A start line: can be request or status line
  - **GET /hello.htm HTTP/1.1 (e.g. of request line from client)**
  - **HTTP/1.1 200 OK (e.g. of status line from server)**
- Zero or more header fields followed by CRLF
  - Provide information about the request or response, or about the object sent in the message body
  - Format for message-header = field-name ":" [field-value]
  - Examples:
    - » **Host: www.example.com (Host required for requests in Ver 1.1)**
    - » **Server: Apache**
    - » **Content-Length: 51**
- An empty line indicating the end of the header fields
- Optionally a message-body
  - If present, carries the actual data
    - <html> <body> <h1>Hello, World!</h1> </body> </html>

HTTP Methods

- GET: Used to retrieve information from the given server using a given URI.
  - should only retrieve data and should have no other effect on the data.
- POST: Used to send data to the server, e.g. customer info, using HTML forms.
- Other methods: PUT, DELETE, TRACE etc

Links

- *Link*: Some text or image you can click to jump to another document or a specific part of the current document.
  - *<a>*: element for links (internal and external).
  - *href*: A required attribute that specifies the destination address
  - *Link text*: The visible part.
    - Click on link text sends you to the specified address.

    Link text
  - You can also use an image as a link.

```
<a href="default.html">
<img src="smiley.gif" alt="HTML tutorial"
        style="width:42px;height:42px;border:0">
</a>
```

HTML Forms

- HTML forms are used to collect user input.
  - The **<form>** tag is used to create an HTML form.
  - HTML forms contain **form elements**.
  - The **<input>** element is the most important **form element**.
    - has many variations, depending on the **type** attribute.
  - *Text* Defines normal text input
    - Default width is 20 characters.
  - *Radio* Defines radio button input (for selecting one of many choices)
  - *Submit* Defines a submit button (for submitting the form)

```
<form action="/url_for_processing/" method="post" >
Username: <input type="text" name="username"><br>
<input type="radio" name="gender" value="male" >Male<br>
<input type="radio" name="gender" value="female" >Female<br>
  <input type="submit" value="Submit now" >
  </form>
```

- Other elements: Reset button, Textarea, Checkbox. Dropdown list etc

Web Framework

- *Web framework*: a software framework designed to support development of dynamic websites and services.
  - Alleviate overhead with associated activities
- Frameworks standardize the 'boilerplate' parts.
  - Provide pre-built components so you can focus on unique parts of your project.
  - Repetitive parts handled by framework.
  - Code you use will be well tested, and have less bugs than what you write from scratch.
  - Enforce good development practices.
  - Security features (login, sessions etc) often better implemented in frameworks.
- Limitations:
  - May restrict you in terms of coding paradigms.
  - Steep learning curve.

Which Framework?

- Many different frameworks are available:
  - ASP.NET using C#, Struts in J2EE, Ruby on Rails, other frameworks using PHP, Perl etc.
- Django is a high-level Python Web framework
  - Encourages rapid development and clean, pragmatic design.
  - Build high-performing, elegant Web applications quickly.
  - Adhere to DRY (Don't Repeat Yourself) principle.

Django Framework

- Web framework for perfectionists with deadlines
  - Main focus
    - Dynamic and database driven websites
  - Automate repetitive stuff
  - Rapid development
  - Follow best practices
  - Free
  - Easy to learn
  - Powerful
- Powerful object-relational mapper (ORM)
  - Data models defined entirely in Python
- Automatic admin interface
  - Eliminates tedious work of creating interfaces to add and update content.
- Elegant URL design
  - Flexible, cruft-free URLs
- Template system
  - powerful, extensible template language to separate design, content and Python code

Sites Using Django

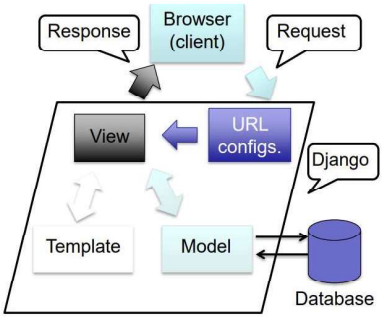
- Disqus
- Instagram
- Mozilla
- NASA
- National Geographic
- OpenStack
- Pinterest

MVC Paradigm

- MVC (Model-View-Controller) paradigm: The application is separated into 3 main layers.
  - Model: Deals with the data
  - View: Defines how to display data
  - Controller: Mediates between the two, allows user to request and manipulate data.
- Allows code reuse
- Increases flexibility
  - E.g. single set of data can be displayed in multiple formats.

Django's MTV Architecture

- MVC → MTV
- *Model*:
  - Deals with data representation/access.
- *Template*:
  - Describes **how** data is represented.
  - Same as 'view' in MVC
- *View*:
  - Describes **which** data is presented.
  - Same as 'controller' in MVC.



Basic Structure

- *DOCTYPE*: Tells browsers *how* to read your document.
  - Forces browsers to use 'standard mode'.
  - Using standard mode, most browsers will read your document the same way.
- *<head>*: Contains information about your page.
- *<body>*: The actual content of your page.

```
<!DOCTYPE html>
<html>
  <head>
      <title>My first Webpage</title>
  </head>
  <body>
      <h1>This is a Heading</h1>
      <p>Hello World!</p>
  </body>
<html>
```

Elements

- HTML elements are marked up using start tags and end tags.
  - Tags are delimited using angle brackets with the tag name in between.
    - End tags include a slash before the tag name.
    - Some elements require only a single tag, e.g.
, <img>
  - HTML tag names are case insensitive.
    - Recommended: use lowercase.
  - Most elements contain some content
    - e.g. ...
  - Elements may contain attributes
    - Used to set various properties of an element.

Attributes

- Attributes: provide additional information about the specific element
  - Always specified in the opening tag.
  - The pattern for writing attributes: attribute="value".
  - Examples:
    - This is tag content
    - <img src="my-pic.jpg" alt="This is a picture ">
    - ...

Headings

- Headings are defined with the <h1> to <h6> elements.
  - Web browsers interpret the headings as the structure of your page.
    - Therefore headings should only be used for actual headings and not for making text bold or large.
    - Headings also increase the readability of your pages.

```
<!DOCTYPE html>
<html>
<head>
    <title> My Webpage</title>
</head>
<body>
    <h1>My Main Heading</h1>
        <p> My first paragraph of the page.</p>
    <h2>Secondary heading</h2>
        <p> My second paragraph.</p>
</body>
</html>
```

Links

- *Link*: Some text or image you can click to jump to another document or a specific part of the current document.
  - <a>: element for links (internal and external).
  - *href*: A required attribute that specifies the destination address
  - *Link text*: The visible part.
    - Click on link text sends you to the specified address.

```
<a href="http://www.mypage.com">Link text</a>
```

  - You can also use an image as a link.

```
<a href="default.html">
<img src="smiley.gif" alt="HTML tutorial" ></a>
```

IDs and Classes

- *ID*: An attribute that assigns a name to your element.
  - The name must be unique and cannot be used anywhere else in your document.
- *Class*: An attribute used to assign some general properties to your element.
  - You can have several elements in your document with the same class-name.
  - You do this, because you want them to behave and look the same way.
- IDs and classes are normally used in combination with CSS

Local Links

- A local link (link to the same web site) is specified with a relative URL.
  - HTML Images
- Link to a specific part of the same doc:
  - Use in conjunction with id or name attribute.
  - # indicates it is a fragment identifier.

```
<p id="para1">This is my first section</p>
<p name="my-para2" >This is another section</p>
<a href="#para1">Go to the first section</a>
<a href="#my-para2">Go to the second section</a>
```

Lists

- HTML has 2 types of lists:
  - Unordered (bulleted) lists: Use <ul> tags
    - list-style-type attribute used to change appearance of bullets. E.g. <ul style="list-style-type:square">
  - Ordered (numbered) lists: Use <ol> tags
    - type attribute used to define type of marker. <ol type="a">
  - Each item in a list is marked by <li> tag.
  - Lists are useful for menus, navigation etc.

```
<ol type="a">
   <li>Math</li>
   <li>History</li>
   <li>English</li>
</ol>
```

Example

```
<body>
<header>
   <h1>My Courses</h1>
</header>
<nav>
      Math<br>
      History<br>
      English<br>
</nav>
<section>
     <h1>Math</h1>
     <p>Math lectures are on Mondays.</p>
</section>
<footer>Copyright © W3Schools.com</footer>
</body>
</html>
```

* Example taken from [1]

HTML Forms

- HTML forms are used to collect user input.
  - The **<form>** tag is used to create an HTML form.

```
<form>
     Form elements
</form>
```

  - HTML forms contain **form elements**.
  - The **<input>** element is the most important **form element**.
    - The <input> element has many variations, depending on the **type** attribute.

Common <input> Elements

- *text* Defines normal text input
  - Default width is 20 characters.
- *radio* Defines radio button input (for selecting one of many choices)
- *submit* Defines a submit button (for submitting the form)

```
<form>
First name: <input type="text" name="firstname"><br>
Last name: <input type="text" name="lastname"><br>
<input type="submit" value="Submit now" >
</form>
```

Radio Button

- **<input type="radio">** defines a **radio button**.
  - Radio buttons let a user select ONE of a limited number of choices:

```
<form>
<input type="radio" name="gender" value="male" checked>
Male
<br>
<input type="radio" name="gender" value="female">Female
</form>
```

Submit Button

- **<input type="submit">** defines a button for **submitting** a form to a **form-handler**.
  - The form-handler is typically a server page with a script for processing input data.
  - The form-handler (url) is specified in the form's **action** attribute.
  - The **method** attribute specifies the HTTP method (**GET** or **POST**) to be used when submitting the forms.

```
<form action="register_page" method="GET">
First name: <input type="text" name="firstname" value="Bob">
<br>
Last name:<input type="text" name="lastname" value="Smith">
<br>
<input type="submit" value="Submit Now">
</form>
```

Reset Buttons

- *reset button*: is used to clear all inputs by the user
  - Example: <input type="reset" value="Clear All" >

Textarea Element

- *<textarea>*: Allows user to enter multiple lines of text – not just a few characters.
- Change the size of you textarea using the attributes: rows and cols.

```
<form method="post">
  <textarea cols="25" rows="7"></textarea>
  <br>
  <input type="submit" value="Submit now">
</form>
```

<label> Tag

- You can associate a label with an element.
  - Label associated with id of the element.
  - Label can be placed before or after the element.

```
<form method="post">
  <label for="comments">Additional Info
  (optional): </label><br>
  <textarea cols="25" rows="7"
  id="comments"></textarea>
  <br>
  <input type="submit" value="Submit now">
</form>
```

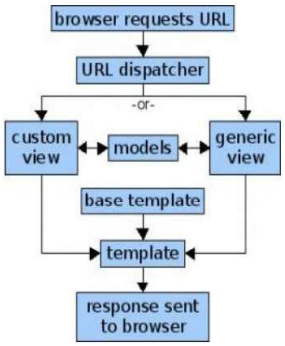
Chapter 3 - Django Models

Django Models

- Topics
  - Creating simple models
    - Rich field types
    - Model inheritance
    - Meta inner class
  - Relationships between models
    - ForeignKey
    - ManyToManyField
  - Advanced usage
  - Getting a model's data
    - Querysets

Review MTV Architecture

- ***Model***
  - Represent data organization; defines a table in a database.
- ***Template***
  - Contain information to be sent to client; help generate final HTML.
- ***View***
  - Actions performed by server to generate data.



Why Use ORM?

- Django provides rich db access layer
  - bridges underlying relational db with Python's object oriented nature
    - ***Portability***: support multiple database backends
    - ***Safety***: less prone to security issues (e.g. SQL injection attacks) arising from malformed or poorly protected query strings.
    - ***Expressiveness***: higher-level query syntax makes it easier to construct complex queries, e.g. by looping over structures.
    - ***Encapsulation***: Easy integration with programming language; ability to define arbitrary instance methods

Defining Models

- ***Model*** is an object that inherits from Model class.
  - Model → represented by a table in the db
  - Field → represented by a table column
- Models are defined and stored in the APP's ***models.py*** file.
- *models.py* is automatically created when you start the APP
  - Contains one line from django.db import models
  - This allows you to import the base model from Django.

Defining Models

- ***Model*** is an object that inherits from Model class.
  - Model → represented by a table in the db
  - Field → represented by a table column
- Models are defined and stored in the APP's ***models.py*** file.
- *models.py* is automatically created when you start the APP
  - Contains one line from django.db import models
  - This allows you to import the base model from Django.


```
File  Edit  Format  Run  Options  Window  Help
from django.db import models

# Create your models here.
class Book(models.Model):
    title = models.CharField(max_length=200)
    length = models.IntegerField()
    website = models.URLField()
    city = models.CharField(max_length=20, blank=True)
    country = models.CharField(max_length=20, default='USA')

    def __str__(self):
        return self.name
```

Field Types

- Django provides a wide range of built-in field types. Some commonly used field types are given below:
  - *CharField*: character string with a limited number of characters.
    - title = models.CharField(**max_length=**200)
  - *TextField*: character string with unlimited number of characters.
  - *IntegerField*: Integer value.
  - *DateTimeField*: contains date as well as time in hours, minutes, and seconds.

null
If **True**, Django will store empty values as **NULL** in the database. Default is **False**.

blank
If **True**, the field is allowed to be blank. Default is **False**.

Field Types

- Django field types (continued):
  - *DateField*: contains the date only.
  - *BooleanField*: stores True or False values.
  - *NullBooleanField*: Similar to above, but allows empty or null value to specify you don't know yet.
  - *FileField*: stores a file path in the db; provides capability to upload a file and store on server.
  - *EmailField*, *URLField*, *IPAddressField*: stored in db like CharField;
    - has extra validation code to ensure value corresponds to valid email, URL, or IP address.

Primary Keys

- Primary key: A field guaranteed to be unique across the entire table.
  - In ORM terms: unique across the entire model.
  - Using auto-incrementing integers for this field is an effective way of ensuring uniqueness.
  - Useful as reference points for relationships between models.
- By default Django automatically creates this field (of type AutoField)

Example

```
class Company(models.Model):
    co_name = models.CharField(max_length=50)

class Car(models.Model):
    type = models.Charfield(max_length=20)
    company = models.ForeignKey(Company,
                on_delete=models.CASCADE)
```

NOTE: The class being referred to must be already defined; otherwise, the variable name would not be available for the Car class.

Example

Alternatively, use a string
– class name if it is defined in same file, or
– dot notation (e.g. 'myApp.Company') if defined in another file

```
class Car(models.Model):
    type = models.Charfield(max_length=20)
    company = models.ForeignKey('Company',
     on_delete=models.CASCADE)

class Company(models.Model):
    co_name = models.CharField(max_length=50)
```

Many-to-Many Relationship

- Uses the ManytoManyField.
- Syntax is similar to ForeignKey field.
- Needs to be defined on one side of the relationship only.
  – Django automatically grants necessary methods and attributes to other side.
  – Relationship is symmetrical by default → doesn't matter which side it is defined on.

Example

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    length = models.IntegerField()
    pub_date = models.DateField()

class Author(models.Model):
    name = models.CharField(max_length=50)
    books = models.ManyToManyField(Book)
```

NOTE: The many-to-many relation is only defined in one model.

Constraining Relationships

- Both *ForeignKey* and *ManyToManyField* take a limit_choices_to argument.
  – takes a dictionary as its value
  – dictionary key-value pairs are query keywords and values
  – powerful tool for defining the possible values of the relationship being defined.

```
class Employee(models.Model):
...
staff_member = models.ForeignKey(User, on_delete=models.CASCADE,
                                 limit_choices_to={'is_staff': True})
```

One-To-One Relationship

- Uses the one-to-one field
  – Requires a positional argument:
  – the class to which the model is related.
  – Useful when an object "extends" another object in some way
- Explicitly defined on only one side of the relationship.
  – The receiving end is able to follow the relationship backward.
- Model inheritance involves an implicit one-to-one relation.

Model Inheritance

- Models can inherit from one another, similar to regular Python classes.
- Previously defined Empolyee class

```
class Employee(models.Model):
    name = models.CharField(max_length=50)
    age = models.IntegerField()
    email = models.EmailField(max_length=100)
    start_date = models. DateField()
```

- Suppose there are 2 types of employees
  – programmers and supervisors

Model Inheritance

- Option 1: Create 2 different models
  – duplicate all common fields
  – violates DRY principle.
- Option 2: Inherit from **Employee** class

```
class Supervisor(Employee):
    dept = models.CharField(max_length=50)

class Programmer(Employee):
    boss = models. ForeignKey(Supervisor,
                on_delete=models.CASCADE)
```

Adding Methods to Models

- Since a model is represented as a class, it can have *attributes* and *methods*.
- One useful method is the __str__ method
  – It controls how the object will be displayed.

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    length = models.IntegerField()
    pub_date = models.DateField()

    def __str__(self):
        return self.title
```

Meta Inner Class

- ***Meta class***: Used to inform Django of various metadata about the model.
  – E.g. display options, ordering, multi-field uniqueness etc.

```
class Employee(models.Model):
    name = models.CharField(max_length=50)
    age = models.IntegerField()
    email = models.EmailField(max_length=100)
    start_date = models.DateField()

    class Meta:
        ordering = ['name', 'start_date']
        unique_together = ['name', 'age']
```

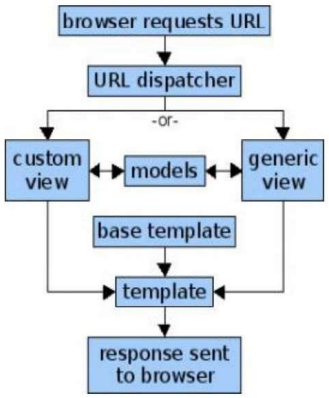
Chapter 5 – Django Views

Review MTV Architecture

Django Views

- Topics
  - URLS
  - HTTP Objects
    - Request
    - Response
  - Views
    - Custom views
    - Generic views (if time permits)

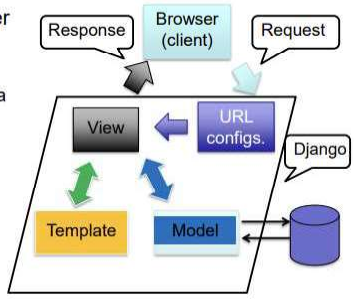
- Represent data organization; defines a table in a database.
- Contain information to be sent to client; help generate final HTML.
- Actions performed by server to generate data.



www.tikalk.com/files/intro-to-**django**.ppt

Choosing a View (Function)

- Django web pages and other content are delivered by views.
  - Each view is represented by a simple Python function (or method)
- Django chooses a view by examining the requested URL
  - Only looks at the part of URL after the domain name.
  - Chooses view that 'matches' associated URL pattern.



URLconf

- *URLconf (URL configuration): maps between URL path expressions to Python functions (your views).*
- *urlpatterns*: a sequence of Django **paths**

  Example: **path(r'', views.index, name='index'),**
- URL patterns for your app/project specified in corresponding ***urls.py*** file.

Sample urls.py

mysite/urls.py

```
from django.urls import include, path
from django.contrib import admin
urlpatterns = [
  path(r'admin/', admin.site.urls),
  path(r'myapp/',
  include('myapp.urls')),
]
```

myapp/urls.py

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

app_name = 'myapp'
urlpatterns = [
  path(r'', views.index, name='index'),
]
```

- include(*module*, *namespace=None*)
  - urlpatterns can "include" other URLconf modules.
  - This "roots" a set of URLs below other ones
  - When Django encounters include():
    - it chops off part of the URL matched up to that point
    - sends the remaining string to the included URLconf for further processing
  - Always use include() when including other URL patterns
    - Only exception in admin.site.urls

path()

- Syntax:
  - path(route, view, kwargs=None, name=None)
  - ***route***: a string that contains a URL pattern
    - may contain angle brackets (like **<username>**) to capture part of the URL and send it as a keyword argument to the view.
    - angle brackets may include a converter specification (like the int part of <int:section>) which limits the characters matched and may also change the type of the variable passed to the view
    - Django starts at the first **path**, compares requested URL against each route until it finds one that matches.
      - Does not search GET and POST parameters, or domain name

path()

- Syntax:
  - **path(route, view, kwargs=None, name=None)**
  - ***view***: after finding match, Django calls specified view function, with
    - HttpRequest object as the first argument and
    - any "captured" values from the regular expression as other arguments.
  - ***kwargs*** : can pass additional arguments in a dict, to view function
  - ***name***: lets you refer to URL unambiguously from elsewhere in Django

path()

- Examples:

```
from django.urls import include, path

urlpatterns = [
  path('index/', views.index, name='main-view'),
  path('bio/<username>/', views.bio, name='bio'),
  path('articles/<slug:title>/', views.article,
        name='article-detail'),
  path('articles/<slug:title>/<int:section>/', views.section,
        name='article-section'),
  path('weblog/', include('blog.urls')),
  ...
]
```

* A **slug** is a short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs.

Response Objects

- Setting HTTP headers:
  - Treat response object as a dictionary.
  - 'key/value' pairs correspond to different headers and corresponding values.
    - HTTP header fields cannot contain newlines.
  - Example:

```
response = HttpResponse()
response["Content-Type"] = "text/csv"
response["Content-Length"] = 256
```

View Functions

```
def index(request):
    books = Book.objects.all() [:10]
    response = HttpResponse()
    heading1 = '<p>' + 'List of books: ' + '</p>'
    response.write(heading1)
    for book in books:
        para = '<p>' + str(book.id) + ':  ' + str(book) + '</p>'
        response.write(para)
    return response
```

View Functions

```
def about(request):
    return HttpResponse('Sample Website')

def detail(request, book_id):
    book = Book.objects.get(id=book_id)
    response = HttpResponse()
    title = '<p>' + book.title+ '</p>'
    author = '<p>' + str(book.author)+ '</p>'
    response.write(title)
    response.write(author)
    return response
```

HttpResponse Subclasses

- Django provides HttpResponse subclasses for common response types.
  - *HttpResponseForbidden*: uses HTTP 403 status code
  - *HttpResponseServerError*: for HTTP 500 or internal server errors
  - *HttpResponseRedirect*: the path to redirect to (required 1st agrument to the constructor)
  - *HttpResponseBadRequest*: acts like HttpResponse, but uses a 400 status code
  - *HttpResponseNotFound*: acts like HttpResponse, but uses a 404 status code

Common View Operations

1. ***CRUD***: **C**reate, **R**ead (or Retrieve), **U**pdate and **D**elete
2. Load a template
3. Fill a context
4. Return HttpResponse object
   - with result of the rendered template

Summary

- Choosing a view
  - URLs, patterns
- Request and response objects
  - HttpResponse subclasses
- Views
  - Common operations - CRUD

Tags

- Tags {% tag %} can have different functionality.
  - e.g. control flow, loops, logic
  - may require beginning and ending tags
  - some useful tags:
    - for
    - if, elif, else
    - block and extends

for Tag

- Used to loop over each item in an array
- Example:

```
<ul>
{% for book in booklist %}
    <li>{{ book.title }}</li>
{% endfor %}
</ul>
```

if, elif, else Tags

- Evaluates a variable
  - if that variable is "true" the contents of the block are displayed.

```
{% if my_list|length > 5 %}
  Number of selected items: {{ my_list|length }}
{% elif my_list %}
  Only a few items were selected
{% else %}
  {{my_list|default: 'Nothing selected.'}}
{% endif %}
```

url Tag

- *url Tag*: Returns an absolute path reference (a URL without the domain name) matching a given view function.
  - may have optional parameters v1 v2 etc
    - Do not mix both positional and keyword syntax in a single call.
  - All arguments required by the URLconf should be present.
    - **{% url** 'path.to.some_view' v1 v2 **%}**
    - **{% url** 'path.to.some_view' arg1=v1 arg2=v2 **%}**

Removing Hardcoded URLs

```
urlpatterns = [
  path(r'<int:emp_id>/', views.detail, name='detail'),
]
```

Matching url: **myapp/5/**

A hardcoded link in template file:

- {{ emp.name }}
  - hard to change URLs on projects with many templates
  - Solution: use the {% url %} template tag, if name argument is defined in the corresponding ***urls.py***
- {{emp.name }}
  - looks up URL definition from the ***myapp.urls*** module
  - **path(r'<int:emp_id>/', views.detail, name='detail')**
- If you want to change the URL
  - Matching url: **myapp/5/ → myapp/emp_info/5/**
  - **path(r'emp_info/<int:emp_id>/', views.detail, name='detail')**
  - Don't need to change anything in template file

Namespacing URL Names

- Adding namespaces allows Django to distinguish between views with same names in different APPs.
  - add namespace in app level *urls.py* (after import instructions)
    - app_name = 'myapp'
  - URL definition from the myapp.urls module
    - path(r'<int:emp_no>/', views1.detail, name='detail')
  - In template file, refer to it as
  - {{emp.name}}
  - Assuming emp.id=2, url tag will output string: /myapp/2 /
  - Final HTML string: {{emp.name}}

```
1  from django.urls import path
2  from myapp import views1
3
4  app_name = 'myapp'
5
6  urlpatterns = [
7      path(r'', views1.index, name='index'),
8      path(r'about/', views1.about,
           name='about'),
9      path(r'<int:emp_no>/',
           views1.detail, name='detail'),
10 ]
```

Template Inheritance

- *Template inheritance*: allows you to build a base "skeleton" template that contains all the common elements of your site.
  - defines **blocks** that child templates can override
  - *block Tag*: Used in base template to define blocks that can be overridden by child templates.
    - tells template engine that a child template may override those portions of the template
    - <title>**{% block** title **%}**Hello World**{% endblock %}**</title>
  - ***extends Tag***: Used in child template
    - tells the template engine that this template "extends" another template.

Base&Child Templates

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{%
      endblock %}</title>
</head>
<body>
    {% block sidebar %}
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/blog/">Blog</a></li>
    </ul>
    {% endblock %}
    {% block content %}{% endblock %}
</body>
  </html>
```

```
{% extends "base.html" %}

{% block title %}My amazing blog{%
   endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

Final HTML

```
{% extends "base.html" %}

{% block title %}My amazing blog{%
   endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>
<body>
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/blog/">Blog</a></li>
    </ul>
    <h2>Entry one</h2>
    <p>This is my first entry.</p>

    <h2>Entry two</h2>
    <p>This is my second entry.</p>
</body>
  </html>
```

include Tag

- *include*: Loads a template and renders it with the current context.
  - template name can either be a variable or a hard-coded (quoted) string, in either single or double quotes.
    - **{% include** template_name **%}**
    - **{% include** "welcome.html" **%}**
    - Context: variable person is set to "John", greeting is set to "Hello"
      - {'greeting': 'Hello', 'person': 'John'}
    - welcome.html template: **{{** greeting **}}**, **{{** person **}}**!
    - **{% include** "welcome.html" **%}** → Hello, John!

Chapter 7 – 8 : Forms

Review MTV Architecture

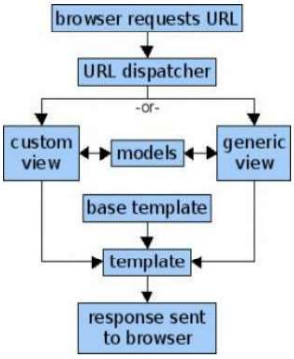
Django Forms

- Topics
  - Django Forms
    - The Form Class
    - Fields and Widgets
  - Rendering Forms
    - Validating Forms
  - ModelForm

– Represent data organization; defines a table in a database.

– Contain information to be sent to client; help generate final HTML.

– Actions performed by server to generate data.



HTML Forms

- *Form*: A collection of elements inside <form>...</form>
  - allow user to enter text, select options, manipulate objects etc.
  - send information back to the server.
- In addition to <input> elements, a form must specify:
  - *where*: the URL to which the data corresponding to the user's input should be returned
  - *how*: the HTTP method to be used to return data.
    - **<form action="/your-name/" method="post">**

GET and POST

- ***GET***: bundles the submitted data into a string, and uses this to compose a URL.
  - The URL contains the address where the data must be sent, as well as the data keys and values.
- ***POST***: Form data is transmitted in body of request, not in URL.
  - Any request that could be used to change the state of the system should use POST method.

GET and POST

- GET should be used **only** for requests that **do not** affect the state of the system.
  - Not suitable for large quantities of data, or for binary data, such as an image.
  - Unsuitable for a password form, because the password would appear in the URL.
  - GET is suitable for things like a web search form
    - the URLs that represent a GET request can easily be bookmarked, shared, or resubmitted

Django's Functionality

- Form processing involves many tasks:
  - Example: prepare data for display, render HTML, validate data, and save data
- Django can automate and simplify much of this work. Handles 3 main areas:
  - preparing and restructuring data ready for rendering
  - creating HTML forms for the data
  - receiving and processing submitted forms and data from the client

Building a Form

- Sample HTML form, to input your name.

```
<form action="/inp/" method="post">
    <label for="your_name">Username: </label>
    <input id="your_name" type="text" name="your_name"
        maxlength="100" >
    <input type="submit" value="OK">
</form>
```

- Components:
  - Data returned to URL /inp/ using POST
  - Text field labeled Username:
  - Button marked "OK"

Building a Django Form

- Create a Form subclass:

```
from django import forms
class NameForm(forms.Form):
    your_name = forms.CharField(max_length=100)
```

  - This defines a Form class with a single field (your_name).
    - Creates a text input field
    - Associates a label with this field
    - Sets a maximum length of 100 for the input field.
  - When rendered it will create the following HTML

```
<label for="id_your_name">Your name: </label>
<input id="id_your_name" type="text" name="your_name"
    maxlength="100" >
```

- NOTE: It does not include <form> </form> tags or submit button.

The Form Class

- ***Form class***: describes a form and determines how it works and appears.
  - Similar to how a model describes the logical structure of an object
- ***Form Field***: a form class's fields map to HTML form <input> elements.
  - A form's fields are themselves classes;
  - Fields manage form data and perform validation when a form is submitted.
  - A form field is represented in the browser as a HTML "widget"

Rendering Options

```
from django import forms
class ContactForm(forms.Form):
  subject =
  forms.CharField(max_length=100)
  message =
  forms.CharField(widget=forms.Tex
  tarea)
  sender = forms.EmailField()
  cc_myself =
  forms.BooleanField(required=Fals
  e)
```

- The name for each tag is from its attribute name.
  - The text label for each field is generated from the field name by converting all underscores to spaces and upper-casing the first letter. Default suffix is ':'
  - Example: cc_myself → 'Cc myself:'
  - These are defaults; you can also specify labels manually.
- Each text label is surrounded in an HTML <label> tag, which points to a form field via its id.
  - Its id is generated by prepending 'id_' to the field name.
- The id attributes and <label> tags are included in the output by default.
  - To change this, **set auto_id=False**

Rendering Forms

```
from django import forms
Class ContactForm(forms.Form):
  subject =
  forms.CharField(max_length=1
  00)
  message =
  forms.CharField(widget=forms.
  Textarea)
  sender = forms.EmailField()
  cc_myself =
  forms.BooleanField(required=F
  alse)
```

- Output of {{myform.as_p}}

```
<form action="/myapp/contact/" method="post">
 <p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject"
           maxlength="100" /></p>
 <p><label for="id_message">Message:</label>
    <input type="text" name="message" id=
           "id_message" /></p>
 <p><label for="id_sender">Sender:</label>
    <input type="email" name="sender"
           id="id_sender"/></p>
 <p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself"
           id="id_cc_myself" /></p>
<input type="submit" value="Enter Contact Info" />
</form>
```

Form Validation

- ***Form.is_valid()*** : A method used to validate form data.
  - bound form: runs validation and returns a boolean (**True** or **False**) designating whether the data was valid. Generates myform.errors attribute.
  - unbound form: always returns **False**; myform.errors = { }
- The validated form data will be in the myform.cleaned_data dictionary.
  - includes a key-value for ***all*** fields; even if the data didn't include a value for some optional fields.
  - Data converted to appropriate Python types
    - Example: IntegerField and FloatField convert values to Python int and float respectively.

Validated Field Data

```
>>> data = {'subject': 'hello',
            'message': 'Hi there',
            'sender': 'foo@example.com',
            'cc_myself': True}
>>> myform = ContactForm(data)
>>> myform.is_valid()
True
>>> myform.cleaned_data
{'cc_myself': True,
   'message': u'Hi there',
   'sender': u'foo@example.com',
   'subject': u'hello'}
```

- The values in **cleaned_data** can be assigned to variables and used in the view function.

```
if myform.is_valid():
  subj= myform.cleaned_data['subject']
  msg= myform.cleaned_data['message']
  sender = myform.cleaned_data['sender']
  cc = myform.cleaned_data['cc_myself']

  return HttpResponseRedirect('/thanks/')
```

Form Validation – if Errors Found

Django automatically displays suitable error messages.

- ***f.errors*****:** An attribute consisting of a ***dict*** of error messages.

form's data validated first time either you call is_valid() or access errors attribute.

- ***f.non_field_errors()*****:** A method that returns the list of errors from f.errors not associated with a particular field.
- ***f.name_of_field.errors***: a list of form errors for a specific field, rendered as an unordered list.

E.g. form.sender.errors() → [u'Enter a valid email address.']

Displaying Errors

- Rendering a *bound* **Form** object automatically runs the form's validation
  - HTML output includes validation errors as a <ul class="errorlist"> near the field.
  - The particular positioning of the error messages depends on the output method.

```
>>> data = {'subject': '', 'message': 'Hi there', 'sender': 'invalid email format',
                           'cc_myself': True}
>>> f = ContactForm(data, auto_id=False)
>>> print(f.as_p ())


<p><ul class="errorlist"><li>This field is required.</li></ul></p>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
 <p><ul class="errorlist"><li>Enter a valid email address.</li></ul></p>
<p>Sender: <input type="email" name="sender" value="invalid email address"
   /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself"
   /></p>
```

ModelForm

- ***ModelForm*****:** a helper class to create a **Form** class from a Django **Model**.
  - The generated Form class will have a form field for every model field
    - the order specified in the fields attribute.
  - Each model field has a corresponding default form field.
    - Example: **CharField** on model → **CharField** on form.
    - **ForeignKey** represented by **ModelChoiceFeild:** a ChoiceField whose choices are a model QuerySet.
    - **ManyToManyFeild** represented by **ModelMultipleChoiceField:** a MultipleChoiceField whose choices are a model QuerySet.
    - If the model field has blank=True, then required = False .
    - The field's label is set to the verbose_name of the model field, with the first character capitalized.
    - If the model field has choices set, then the form field's widget will be set to Select, with choices coming from the model field's choices.

ModelForm Example

```
class Book(models.Model):
   title = models.CharField(max_length=100)
   length = models.IntegerField()
   pub_date = models.DateField()

from django.forms import ModelForm
from myapp.models import  Book

# Create the form class.
class BookForm(ModelForm):
      class Meta:
            model = Book
            fields = ['title', 'pub_date', 'length']

# Creating a form to add a book
form = BookForm()

# Create form to change book in db.
book = Book.objects.get(pk=1)
form = BookForm(instance=book)
```

Django Authentication

- Topics
  - Introduction
  - Web Authentication
    - Login/Logout
    - Limiting access
  - Permissions and Authorization
    - Groups
    - Default permissions

Authentication System

- Django's authentication system consists of:
  - ***User*** objects
  - A configurable password hashing system
  - Forms and view tools for logging in users, or restricting content.
  - **Permissions:** Binary (yes/no) flags designating whether a user may perform a certain task.
  - **Groups**: A generic way of applying labels and permissions to more than one user.

Installation

- Add these 2 items in INSTALLED_APPS setting:
  - 'django.contrib.auth' : contains the core of the authentication framework, and its default models.
  - 'django.contrib.contenttypes': allows permissions to be associated with models you create.
- Add these 2 items in MIDDLEWARE_CLASSES setting:
  - SessionMiddleware: manages sessions across requests.
  - AuthenticationMiddleware: associates users with requests using sessions.
- By default: already included in settings.py.

User Objects

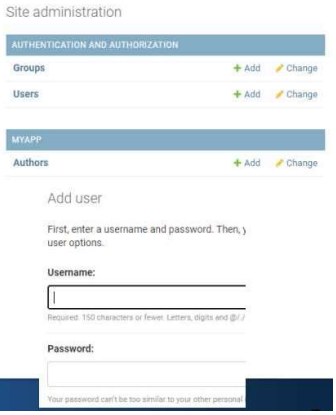
- User objects are the core of the authentication system.
  - Typically represent people interacting with your site.
  - Used to enable things like restricting access, registering user profiles etc.
  - Only one class of user exists in Django's authentication framework
    - Different user types e.g. 'superusers' or admin 'staff' users are just user objects with special attributes set
      - not different classes of user objects.

User Attributes

- The primary attributes of the default user are:
  - ***Username***: Required. 30 characters or fewer.
    - May contain alphanumeric, _, @, +, . and - characters.
  - *first_name*: Optional. 30 characters or fewer.
  - *last_name*: Optional. 30 characters or fewer.
  - *Email*: Optional. Email address.
  - ***Password***: Required.
    - A hash of, and metadata about, the password.
    - Django doesn't store the raw password. Raw passwords can be arbitrarily long and can contain any character.

Using Admin Interface

- Admin module can be used to view and manage users, groups, and permissions.
  - Both django.contrib.admin and django.contrib.auth must be installed.
  - The "Add user" admin page requires you to choose a username and password before allowing you to edit the rest of the user's fields.
  - User passwords are **not** displayed in the admin (nor stored in the database).
    - a link to a password change form allows admins to change user passwords



Creating Users

- Use create_user() helper function:

```
from django.contrib.auth.models import User
user = User.objects.create_user('john',
  'lennon@thebeatles.com', 'johnpassword')
# At this point, user is a User object that has
# already been saved to the database. You can
# continue to change its attributes.
user.last_name = 'Lennon'
 user.save()
```

Changing Passwords

- Django does not store raw (clear text) passwords on the user model,
  - It only stores a hash.
  - Do **not** manipulate the password attribute of the user directly.
    - **user.password = 'new password' # Don't do this!**
  - Passwords can be changed using **set_password()**

```
from django.contrib.auth.models import User
u = User.objects.get(username='john')
u.set_password('new password')
u.save()
```

Check and Change User Permissions

- User objects have many-to-many field: **user_permissions**

```
myuser = User.objects.get(pk=1)
myuser.user_permissions.set([permission_list])
myuser.user_permissions.add(perm1, perm2, …)
myuser.user_permissions.remove(perm1, perm2, ...)
myuser.user_permissions.clear()
```

Groups

- Groups allow you to apply permissions to a group of users.
  - A user in a group automatically has the permissions granted to that group.
  - Also a convenient way to categorize users to give them some label, or extended functionality.
- User objects have many-to-many field: **groups**

```
myuser = User.objects.get(pk=1)
myuser.groups.set([group_list])
myuser.groups.add(group1, group,2 ...)
myuser.groups.remove(group1, group2, ...)
myuser.groups.clear()
# Add group permissions
group = Group.objects.get(name='wizard')
group.permissions.add(permission)
```

- [1] https://docs.djangoproject.com/en/3.0/topics/auth/

Using Permissions in Your Code

- Restrict certain actions to specific users:
  - Check if user has permission before executing code. Example, if only certain users can serch the library.

```
def searchlib(request):
  if request.user.has_perm('libapp.can_search'):
    # Code to process search request goes here
    #  …
  else:
    return HttpResponse('You do not have permission to search!')
```

Summary

- Authentication
  - User objects
  - Authenticate(), login() and logout()
  - Permissions and Authorization
    - Groups
    - Default permissions

Testing Cookies

1. Call set_test_cookie() method of request.session in a view.
2. Call test_cookie_worked() in a *subsequent* view – NOT in the same view.
   - you can't actually tell whether a browser accepted it until the browser's *next* request.
3. It's good practice to use delete_test_cookie() to clean up afterwards.
   - Do this after you've verified that the test cookie worked.

```
def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try again.")
    request.session.set_test_cookie()
    return render('foo/login_form.html')
```

Sessions Outside of Views

- An API is available to manipulate session data outside of a view.
  - The SessionStore object can be imported directly from the appropriate backend.
  - For django.contrib.sessions.models each session is a normal Django model.
  - Can be accessed using normal Django db API.

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2015, 8, 20, 13, 35, 12)
```

Session Expiration

- *SESSION_EXPIRE_AT_BROWSER_CLOSE*:
  - Controls if session framework uses browser-length sessions or persistent sessions.
  - Global default setting for session framework
  - ***get_expire_at_browser_close()***: True if session cookie expires when user's browser is closed.
    - Can be overwritten at a per-session level by explicitly calling the set_expiry() method of request.session.
  - By default, it is set to False
    - This means session cookies will be stored in users' browsers for as long as SESSION_COOKIE_AGE.
    - Use this if you don't want people to have to log in every time they open a browser
  - If it is set to True
    - Cookies expire as soon as user closes their browser.
    - Use this if you want people to have to **log in every time** they open a browser

Clearing SessionStore

- When a user logs in, Django adds a row to the django_session db table.
  - Django updates this row each time the session data changes.
  - If the user logs out manually, Django deletes the row.
  - If the user does *not* log out, the row never gets deleted.
- As users create new sessions on your website, session data can accumulate in your session store.
  - If you're using the database backend, the django_session db table will grow.
  - If you're using the file backend, your temporary directory will contain an increasing no. of files.
- Django does *not* provide automatic purging of expired sessions.
  - It is your job to purge expired sessions on a regular basis.
  - Django provides a clean-up management command for this purpose: clearsessions.
  - It is recommended to call this command on a regular basis.

ModelAdmin Actions

- ***Actions***: simple functions that get called with a list of objects selected on the change list page.
  - Very useful for making same change to many objects at once.
  - The function takes 3 arguments:
    - The current ModelAdmin
    - An HttpRequest representing the current request,
    - A QuerySet containing the set of objects selected by the user.
- Two main steps:
  - Writing actions
  - Adding actions to ModelAdmin
- Example: You want to update status of several articles at once.
  - Relevant models and functions defined in following slides.

Example

```
# Book model
class Book(models.Model):
    STATUS_CHOICES = (
        (0, 'In stock'),
        (1, 'Available soon'),
        (2, 'Not Available'),
    )
    title = models.CharField(max_length=100)
    length = models.IntegerField()
    pub_date = models.DateField()
    status =
  models.IntegerField(choices=STATUS_CHOICES)
```

Writing Actions

```
# Book model
class Book(models.Model):
    STATUS_CHOICES = (
        (0, 'In stock'),
        (1, 'Available soon'),
        (2, 'Not Available'),
    )
    title =
models.CharField(max_lengt
h=100)
    length =
models.IntegerField()
    pub_date =
models.DateField()
    status =
models.IntegerField(choices=
STATUS_CHOICES)
```

```
def make_available(modeladmin, request,
  queryset):
      queryset.update(status=0)
```

- Default name in action list → "Make available"
- Also possible to iterate over queryset

```
  for obj in queryset:
      obj.status = 0
```

- Provide friendly description in action list.

```
def make_available (modeladmin, request,
  queryset):
  queryset.update(status=0)
  make_available.short_description = "Mark as
  available"
```

Adding Actions

- To inform our ModelAdmin of the action:
  - Add the action to the list of available actions for the object.
  - "delete selected objects" action available to all models

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

def make_available(modeladmin, request, queryset):
    queryset.update(status=0)
    return
make_available.short_description = 'Mark as available' #

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'status')
    actions = [make_available]

# Register your models here.
admin.site.register(Book, BookAdmin)
```

Fields Option

- Used to make simple changes in the layout of fields in the forms.
  - showing a subset of the available fields, modifying their order or grouping them in rows
- CAUTION: If a **required** field is excluded, it will cause ERROR when trying to save the object.
  - If only blank=True in model → ERROR
  - If a default value is provided in the model → OK.

```
class BookAdmin(admin.ModelAdmin):
    fields = ('title', 'length', 'pub_date')
```

  - displays only the above three fields for the model, in the order specified.
  - To show multiple fields on the same line, wrap those fields in their own tuple

```
class BookAdmin(admin.ModelAdmin):
    fields = ('title', ('length', 'pub_date'))
```

Inlines

- ***Inlines***: Provides admin interface the ability to edit models on the same page as a parent model.
  - Two Subclasses: TabularInline and StackedInline
    - The difference between these two is merely the template used to render them
  - To edit the cars made by a company on the company page: Add inlines to a model

```
# Suppose two models are defined
from django.db import models

class Company(models.Model):
    co_name =
  models.CharField(max_length=50)

class Car(models.Model):
    type =
  models.CharField(max_length=20)
    company =
  models.ForeignKey(Company)
```

```
from django.contrib import admin

class CarInline(admin.TabularInline):
    model = Car

class
CompanyAdmin(admin.ModelAdmin):
  inlines = [
      CarInline,
  ]
```

Summary

- Admin Site
- ModelAdmin Objects
  - Registering objects
- Options and methods
  - Actions, fields and fieldsets
- InlineModelAdmin Objects
  - TabularInline
  - StackedInline

- [1] https://docs.djangoproject.com/en//ref/contrib/admin/