

Backend

Command Prompt →

- ① If want to know subfolders in directory
 `dir`
 - android → signifies that folder is hidden
- ② changing directory to go back
 `cd name`
 `cd ..`
- ③ clear
 `cls`
- ④ printing current directory
 `cd`

Python

- ① Number <
 `int`
 `float`
- ② # comment
- ③ `print(my_string[2:])` → grab all letters starting from index 2.
 `my_string[:3]` → prints upto 3rd index excluding 3rd index
 `[2:5]` → all letters from 2 to 5 excluding 5
 `[::-2]` → step size 2
 → prints every 2nd letter
 `[::-1]` → step size -1
 → reverse string
- ④ String is immutable
 `sts[0] = 'x'` → X
- ⑤ `my_string.lower()`
- ⑥ `my_string.split()` → ['Hello', 'World']
 `split('e')` → ['H', 'lloword']

• Print formating →

`x = "Item One": {} Item two : {}".format("dog", "cat")`

o/p Item One: Dog Item Two: Cat

→ DICTON

• List →

`myList = [1, 2, 3]` → `myList[0]` → 1
→ `myList[-1]` → 3

`myList = ['hey', 1, 2, 3, 12, True]`

`print(len(myList))` → length

Add an item → `myList.append("New")`

`myList = [1, 2, 3]` → add to end of list
`listTwo = [4, 5, 6]`

`myList.extend(listTwo)` → [1, 2, 3, 4, 5, 6]

Remove

`item = myList.pop()` → 1 2 3

`print(item)` → 1

`print(myList)` → [2, 3]

`item = myList.pop(1)` → [1] → remove it

Reverse →

`myList.reverse()`

`print(myList)` → [3, 2, 1]

Sort

`myList.sort()`

[1, 2, 3]

[3, 2, 1]

Matrix →

`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`first_col = [row[0] for row in matrix]`

→ for

{1, 4, 7}

to tuple

→ Dictionary

my = {"key1": "value", "key2": "value"}

print(my['key2']) → value

print(my)

→ Boolean

True False

→ Tuples → lists but immutable

tuple t = (1, 2, 3)

t = (1, 1.2, "key")

print(t[0])

(indexing) new from old + n/a

(middle) in list

→ Set → unordered collection

x = set()

print(x) → {1, 2, 0, 1} (new - fra)

x.add(1)

any order

x.add(2)

(same item) bba → no duplicates

x.add(0, 1)

smash + insert another allowed

x.add(1)

(s, 1) bba = llwo

('Hello') string

→ if else →

→ and or used in between with = copy

→ ?? x → indentation

→ if 1 < 2:

print('yes!')

elif (2 < 3):

print('NO!')

else print('ya')

→ for

for item in seq → iterate over anything

print(item)

for tuple → for (tup1, tup2) in mypair
print(tup1)
print(tup2)

→ while →

* comment → `#`

while i < 5:

 print("i: {}".format(i))
 i = i + 1

* out = [num**2 for num in seq]

→ for item in range(10)

 print(item)

→ Function →

syn → def my_func(param1='default'):
 print(param1) → default

my_func()

Eg → def add(num1, num2)
 return num1 + num2

result = add(1, 2)

print('result')

→ filter → filters needed input

def even(num)

 if return num % 2 == 0

evens = filter(even, mylist)

print(list(evens))

→ SCOPE →

Local

Enclosing fn gets local

Global

Built in

def func()

 global x
 x = 1000

effects to global n

or (global, local) set a global v

(global) and

(local) change

OOP →

class DogSample():

① def __init__(self): → refer to itself
necessary

② def __init__(self, breed): → self = self
self.breed = breed

mydog = Dog(breed = "Lab")

print(mydog.breed) → Lab

class Dog()

species = mammal

def __init__(self, breed, name):

self.name = ~~goat~~ name

self.breed = ~~dog~~ name

def area(self): → method of class

return self.name * self.name

* Dog.species

mydog = Dog
(name
"goat")
breed
"Lab")

mydog.name
mydog.breed
mydog.species

Calling another class → inheritance

class Animal:

def __init__(self):

print("Animal created")

def eat(self):

print("Eat")

class Dog(Animal):

def __init__(self):

Animal.__init__(self)

print("Dog created")

mya = Animal() → mya.eat() → Eat
mya.eat() → Eat
mydog = Dog() → dog.eat()

→ Animal created → Dog created → Eat
parent int. → child int. → inherited method

Printing class →

```
class Book: page = 100  
    def __init__(self, title, author):  
        self.title = title  
        self.author = author  
  
    print ← def __str__(self):  
        return "Title: {}, Author: {}".format(self.title,  
                                              self.author)  
  
    def len(self):  
        return self.page
```

b = Book("Python")
print(b)

print(len(b))

Error & Exception →

open('n')

open('file-name', 'r')

open('file-name', 'w')

```
f = open('simple.txt', 'w')
```

```
f.write("Test write to")
```

If error → check for input output error

except IOError: don't need to specify

```
print("could not open file")
```

else

else:

```
print("success")
```

```
f.close()
```

finally: → will work even if exception

```
print("I always work")
```

Regular Expressions →

import re → importing regular exp.

patterns = ['term1', 'term2']

text = 'this is a string with terms but not other'
for pattern in patterns:

print ("I am searching for " + pattern)

match object ← if re.search(pattern, text) → where to search
attribute key to search
print("match")

else no match

→ read

text = "gt term1"

→ write

match = re.search('term1', text)

print(match.start()) → 3

email = username@gmail.com

print(re.split('@', email))

where to split ↴ what to split

[match, 'match'] ← print(re.findall('match', 'test phrase match to match'))

Django

used because
packagers get updated

free & open source
web framework

virtual environment → allows to have ~~multiple~~ installations
of python and other packages on your computer

use virtual env with conda by name package
conda create --name myEnv Django

activate by activate myEnv deactivate by deactivate Env

How to start →

① make a directory for your project

mkdir file-name → make a new folder
cd file-name → go into folder

② Activate your env

activate myDjangoEnv

③ create a project

django-admin startproject name

- you will get various files

① init.py → blank, name tells python that it can be used as package

② settings.py → to store your project settings

③ urls.py → store all URL patterns for your project

④ wsgi.py → Python script that acts as the Web server gateway interface. help to deploy web app to production

⑤ manage.py → associates with many commands as we build our web apps

→ Managing & using manage.py →

① python manage.py runserver

copy the last url on chrome — will lead to first project

② **Migration** → reversible, allows to move database from one design to another.

→ Terminologies →

① Django Project → collection of application & config that when combined together will make up the full web application.

② Django Application → created to perform a particular functionality for your entire web application.

Eg → registration app, polling app etc
They are pluggable. Others can use apps from your project & vice versa.

→ **To create app** →

① Creating a app folder

python manage.py startapp name

② Contains file →

① __init__.py → "

② admin.py → can register your model here which Django will then use with ~~other~~ Django's admin interface

③ apps.py → place application specific config

④ models.py → store application's data model

⑤ test.py → store test fn to test your code

⑥ Migration folder → stores database specific info as it relates to the model

③ Need to tell Django that we created an application

→ setting.py > Installed_APPS

Search in file
Add a string 'file-name' to the installed_apps array
name of app
name of file

④ Creating View → sections of application may
e.g. → sending Hello World →

firstapp > view.py →

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

⑤ To see view = linking view to urls.py

→ first-project > urls.py
→ search for urlpatterns = [
Add → from . import views
urlpatterns = [
path('', views.index, name='index'),
path('admin/', admin.site.urls),
]

If run & will see Hello World this is granted

so created application, create view &
map it urls file to work through url.

→ Mapping URLs → Another way [include()]
import from django.conf.urls
include() → allows to look for a match for
regular expressions and link back to our application's
own urls.py file.
we have manually add in urls.py file

Method → add in urls.py of project
from django.conf.urls import include
url patterns = []
path url('app-name/'), include ('app-name.urls')
] this will be the extension
name in the url

steps

① Go to urls.py of first project

add is to

> {

② Make a new file in app folder as urls.py

add

from django.conf.urls import url

from first_app import views

url patterns = []

path url('', views.index, name='index')

Templates

contains static parts of HTML page.

template tags → injects dynamic content that your Django app views will produce, affecting the final HTML

template variables → inject content into the HTML directly from Django

Creating template →

① Under project (first-project) create a new folder named template.

② In the subfolder of first-project > settings.py
(i) add below BASEDIR → to make it run on
TEMPLATEDIR= os.path.join(BASE_DIR, "template") any operating system

(ii) In TEMPLATES [

'DIRS': [↓] = setting new
Add) TEMPLATE_DIRS

Adding a template →

① In template folder, add file Index.html

② Adding a template tag in body →

↓
tag variablename

③ Connecting variable (insert-me) to our project
and application

in first project > views.py :

```
def index(request):
    mydict = {'insert-me': "Hello I am"}
    return render(request, 'index.html', context=
```

↓
html file to add

my-dict
what to
link

④ To make modular →

in templates folder add a folder named on
application and put index.html in it

change * to first_app/index.html

name of app

Static Files

{ } simple text
injections

% % complex
injections
and logical

- ① Creating Folder
- in project folder add a new folder named static and create a folder in it named images
 - first-project > Static > images
 - add image required in the images folder
- ② first-project > settings.py
- Below base-DIR & Template-DIR add →
join base STATIC-DIR = os.path.join(BASE-DIR, "static")
dir to static
 - at the bottom of page.
below STATIC-URL = 'Static'
add → STATICFILES_DIRS = [STATIC-DIR,]
- ★ (Put static & template in app folder.)

- ③ For loading image through html
- <!DOCTYPE html>
 - { % load static % }
 - <body>
 -
 - alt = "Uh not loaded"

- ④ For adding CSS
- make a folder in static name CSS with full name.css
 - link html with CSS as
 - <link rel="stylesheet" href="{} static "css/nameof.css"/>

Create a table

Models

used to incorporate a database into django project

① ~~File~~ go to model.py in project folder

② → **Code**
from django.db import models

① Table built-in class → inherited parent class

class Topic(models.Model)
each attribute refers to a field or a column name
→ top-name = models.CharField(max-length=264, unique=True)

def __str__(self):
return self.topic_name

② Table
class Webpage(models.Model):

topic = models.ForeignKey(Topic)
name = models.CharField(max-length=264, unique=True)
url = models.URLField(unique=True)

def __str__(self):
return self.name

Primary Key → unique identifier for each row in a table.

Foreign Key → denotes that column connects with a primary key of another table

* Similarly for date → DateField etc

③ Apply following command

→ "python manage.py makemigrations" → migrate database corresponding to model

→ "makemigrations first-app" → register changes to

→ "migrate" → migrate database app one more time

→ "shell" → 1st Method to add contents

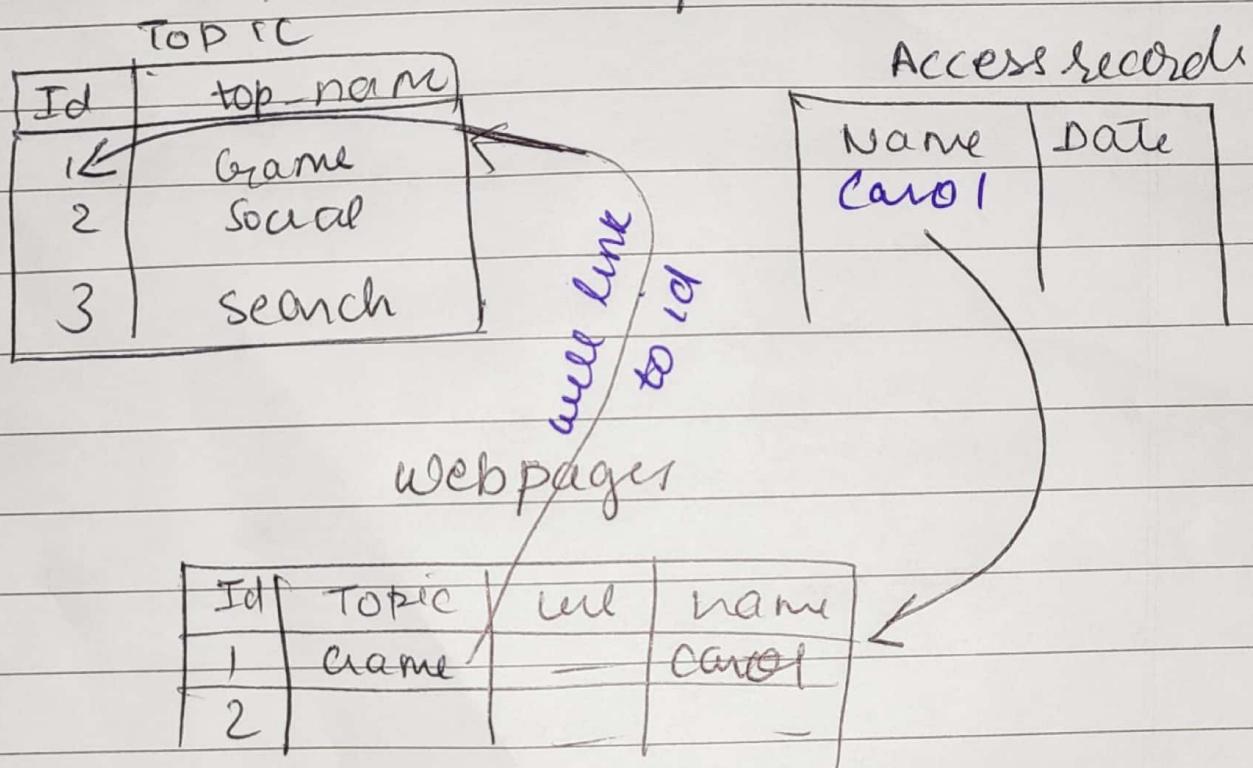
→ from first-app.models import Topic

To save a entry → t = Topic(topic_name="Hy")

t.save()

print(Topic.objects.all())

- * Foreign key links with the id of parameter
we have 3 classes (3 tables)
- 1) Topic → top_name
 - 2) Webpage → Topic (foreign keys), url, name
 - 3) Accessrecord → name (foreign), date



Access Record
Name = model.foreignkey (Webpage)

↳ means

name links to

the webpage table

In name we can input
parameters of webpage table
and get info

against sound design principles, so we instead create a relationship (the 'R' in ORM) between the `yourapp_Course` table and the `yourapp_UserProfile` table.

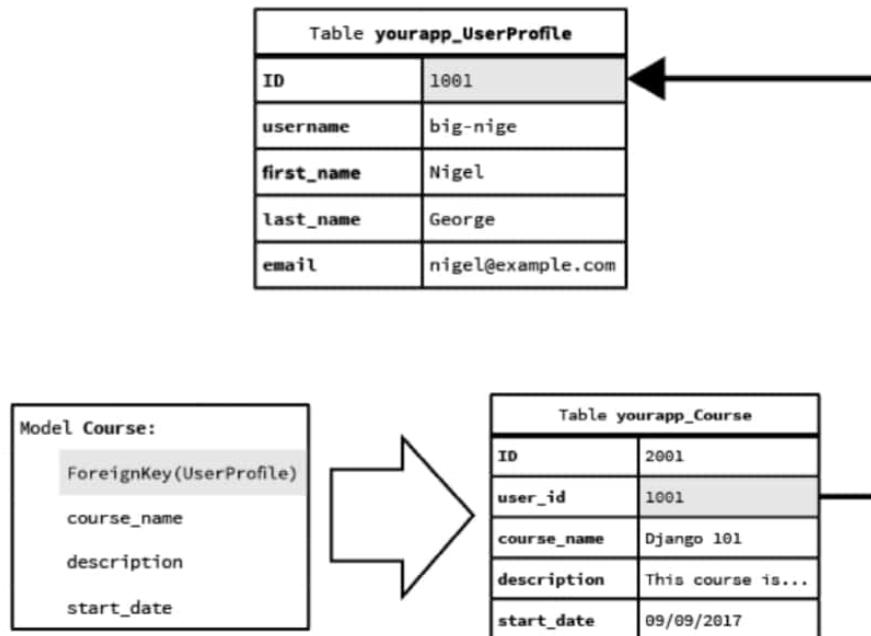


Figure 4-3: Foreign key links in Django models create relationships between tables.

This relationship is created by linking the models with a foreign key—i.e., the `user_id` field in the `yourapp_Course` table is a key field linked to the `id` field in the foreign table `yourapp_UserProfile`.

to quit → quit c)

② Method to add contents

Admin

Method ②

① go to first-app > admin.py

from django.contrib import admin
from first_app.models import Topic, Webpage, AccessRecord

admin.site.register(AccessRecord)

" " " (Topic)

" " " (Webpage)

class name

Topic, Webpage, AccessRecord

register

AccessRecord

Topic, Webpage

classes

② in cmd → to use database and admin, we create a superuser
python manage.py createsuperuser
will ask for username password email
remember to login to admin

→ returns password + username to login to admin

python manage.py runserver

→ site runs server

③ /admin in site leads to → login
leads to a django admin site.

→ Faker library to create script

① Install → cmd → pip install Faker in first project
(code all)

② create populate-first-app.py

③ cmd → " ".py

④ cmd runserver

Models Template Views

① Connecting views to database

Steps →

① Go to views.py

② Add → from first_app.models.import Topic, WebpageAccess
models.py
connecting weewes to database
def index(request):
 webpages_list = AccessRecord.objects.order_by('date')
 date_dict = {'access_record': webpages_list}
 return render(request, 'first_app/index.html', context={
 'date_dict': date_dict})

③ index.html → code
↓
 style as per need → css
 imp → !DOCTYPE
 → { % load static % }
 → <link href="{% static "css/style.css" %}">
 → { % if access-record % } → to check if there
 are access records
 → { % if else % } → else tag
 → { % endif % } → necessary to
 end if
 → { % for acc in access-records % } → for loop
 → { % endfor % } → nece end
 → { % endfor % } → tag

④ runserver

Django forms

Making Django Project from Scratch level II till module

Creating Project Folder, APP folder, my cmd

→ activate mydjangoEnv

→ django-admin startproject ProtoTwo

→ ProtoTwo > python manage.py startapp appTwo

Creating Template → Right click appTwo folder > add folder Template

create → Template > appTwo > index.html > .html

Making Project

Making app

Making appTwo

needed html files

③ Setting.py changes →

① # See template section > Creating template 6 page before

→ TEMPLATE-DIR

→ 'DIRS' = []

Linking & Registering template to Project

② INSTALLED APPS = []

APPNAME
'appTwo'

Registering & linking app to project

④ In

models.py

Refer to

Model section of code

3 page before

→ Create required class

⑤ In views.py
code ->

from AppTwo

appname
import render
import HttpResponse
import class-name

Import

main webpage
duplicate index.html

• def index (request)
return render (request, AppTwo/index.html)

Methods

can similarly insert other methods
linked to respective html files

⑥ urls.py →

① Create file urls.py in AppTwo folder

② In urls.py code → App name

import

{ from django.urls import url
from apptwo import views
urlpatterns [

path ('', views.index, name="index")
path ('', views.users, name="users")

→ main page
→ method called from views.py

③ Go to Project > urls.py

import

{ from django.contrib import admin
from django.urls import path, include
from apptwo import views

urlpatterns [

mainpage ← path ('', views.index, name="index")
admin page ← path ('admin/', admin.site.urls)
users page ← path ('users/', include('apptwo.urls'))
]

⑦ Migration →

protwo > python manage.py migrate → to register all the changes in the app

python manage.py makemigrations AppTwo

python manage.py migrate

⑧ In admin.py →

Check admin section | Method② (3 page before)

⑨ ★ Regularly check for successful changes by
python manage.py summarizer

⑩ Create index.html *

⑪ Populate → Create file in project

Refer laptop ← → code

→ cmd protwo > python file-name

⑫ Create user table in user.html →

⑬ Run server mainpage → index.html
/admin → admin page
/users → user page

If want to create user tab at user page

Django Forms

HTTP → enable communication between a client and a server
for Request/Response
METHODS

Get
request data
from a resource

POST

submits data to be
processed to a resource

Creating basic form →

- ① Create project basicform, app basicapp.
- ② Create template / basic app / index.html
- ③ do changes in settings.py
- ④ create file forms.py in basicapps.
code →
from django import forms
~~from django.core import validators~~

class formName(forms.Form):

name = forms.CharField()

email = forms.EmailField()

~~verify_email = forms.EmailField(label='Enter your mail again!')~~

text = forms.CharField(widget=forms.Textarea)

CLEAN DATA →
All validated & checked data

⑤ Go to views.py

```

from django.shortcuts import render
from . import forms
from forms import formName
import forms from same directory
as current py file
def index(request):
    return render(request, 'basicapp/index.html')
def form_name_view(request):
    form form = forms.formName()
    if request.method == 'POST': → If hit submit
        with method POST
        passing the POST → Method to check if each
        request form = forms.formName(request.POST)
        request → input field has correct
        if form.is_valid() → datatypes
        print("Name" + form.cleaned_data['name']) → dictionary to access valid data
        return render(request, 'basicapp/form-page.html', {'form': form})
    
```

If we get POST back, check if data is valid and so grab data

⑥ Go to urls.py of project

```

from basicapp import views
urlpatterns [
    path('admin/', admin.site.urls),
    path('formpage/', views.form_name_view, name='form-name')
]

```

⑦ Go to form.html

```

<div>
    <form method="POST"> → data to be submitted to be processed
    key from context dictionary {{ form.as_p }} → form to be inside para tag
    MOST → {{ form.csrf_token }} → will accept info and secure it and
    → submit button

```

⑧ answer

check if inputs on form submitted is not going on some other websites and saves from getting false data

Form Validation

① Creating botcatcher → Hidden field remains in HTML but hidden from user. This field is used to catch bots. can be seen in INSPECT.

In formName in forms.py add →

* botcatcher = forms.CharField(required=False, widget=forms.HiddenInput)

creating own Validator

①

OWN VALIDATOR

def clean_botcatcher(self):

keyword checked
to check form item

botcatcher = self.cleaned_data['botcatcher']

dictionary attribute

if length(botcatcher) > 0:

raise forms.ValidationError("caught")

to raise errors

Means a robot scraped the page and filled form and HTML field

* If

check by

bot is caught, will show on page [Hidden field botcatcher caught!]
i.e. if its changes something in HTML

BUILT-IN

from django.core import validators

botcatcher = forms.CharField(

validators=[validators.MaxLengthValidator(0)])

method build-in validator

More validators creating

• def creating a function to check for z

OWN VALIDATOR

check_z(value) must to identify as validator if making own validator
if value[0].lower() != 'z'
raise forms.ValidationError("Name needs to start with z")

add validator name = forms.CharField(validators=[check_z])

→ verify email double check → Method →

add
a variable ← verify_email = forms.EmailField(label="Enter email again")
in class

def clean(self): → keyword to check entire form at once
dictionary ← all_clean_data = super().clean() → grab all clean data at once
email = all_clean_data['email']
vmail = all_clean_data['verify_email']
check for ← if (vmail != email)
match raise forms.ValidationError("Make sure emails match")

Connecting forms to Model

accepting form input and passing it to a model as table

① Frontend

edit index and user + insert form tag in users.html

② Backend

i) forms.py in app Folder

```
from django import forms
from app.models import User
```

class NewUserForm(forms.ModelForm):

If you want validate write form fields here → class Meta():
model = User → link to class provides info connecting the model to the form

fields = '__all__' → for info see page

③ views.py

* a form would directly be generated from User class / model

```
import render
```

only 2 needed

```
from appTwo.forms import NewUserForm
```

def index(request):

def users(request):

form = NewUserForm()

if request.method == "Post":

form = NewUserForm(request.POST)

if form.is_valid():

saved_in admin database & save to form → form.save(commit=True) to database

~~four~~ fields

- ① field = "all" grabs all fields from model
- ② field = ["name", "email"]
exclude those fields and include everything else.
- ③ field = ("field1", "field2")
include these fields. → list of included fields

return index(request) → will take to

use:

print('Error') → homepage

return render(request, 'Apptwo/users.html', {'form': form})

→ Relative URLs with Templates →

i.e. hardcoded urls → href, with URL template

steps → 1. Create basicapp, templates, static folder, media folder, static files

① Create project learning_templates, app basicapp, templates and templates/basicapp/index.html, other.html, base.html, static relative_url_template.html

② Change settings.py

③ views.py

```
def index():
    return —
def other():
    return —
def relative():
    return —
```

④ project > urls.py

```
path('' — — —
      path('admin' — — —
            path('basicapp', include('basicapp.urls')))
```

⑤ basic app > urls.py

app_name = 'basicapp' → for template tagging

```
urlpatterns = [
    path('relative/', view = — — —),
    path('other/', view = — — — name='other'),
```

⑥ in relative_url_template.html:

add → ↵ basicapp:other ↵ Other Page ↵
 ↵ (two systems) ↵ ↵ No space or django will look for -basicapp
 Now it is relative to actual application only

For admin link → already installed
apps

admin

for index link →

— " {% url 'index' %} " —

Template Inheritance

- allows to create base template we can inherit from
- saves repetitive work
- template extending : extending base.html to other html files

Steps →

① In base.html

<links to JS, CSS, — >

<html navbar>

<body>

Everything outside this will be extended to all pages

{% block body-block %}

{% endblock %}

In other.html file

<!DOCTYPE html> → MUST

{% extends "basicapp/base.html" %}

{% block body-block %}

anything

inside will show

{% endblock %}

Template Filters

General form of template filter →

{% value | filter : "parameter" %}

* search django + templates for documentation

① Built-in template filters →
Adding a tag in index →

In views.py

def index —

context_dict = { 'text': 'Hello', 'number': 100 }

return render(— , context_dict)

② adding tag in index.html & adding filter to tag

`{% text|upper %}` → hello → HELLO

`{% number|filter: "gg" %}` → 100 → 188
value parameter

IV

③ For creating own filters →

① → create template tags > `__init__.py` → to act as model
folder file
`my-extras.py` file

② → in `my-extras.py` →

`from django import template`
`register = template.Library()`

a library where
tags and filters
are registered

`@register.filter` → If no arg used
will take fn name as name
`def cutout(value, args)`
`return value.replace(args, "")`

`@register.filter(name='lower')`

`def lower(value)`
`return value.lower`

`def cutout`
`return`

`register.filter`
`'cutout' 'cut'`

`fn name`
python function

③ In `index.html`

`<! DO` →

`g>. extend` →

`{% load my-extras %}` MUST

`g>. body` →

`ch> index</h1>`

`{% text|cutout:'rl' %}` → Hello World → Hello Wod

① `@register.filter`
② `register.filter('name', python fn)`