**INTRODUCTION:**

Hey there, Welcome to the Django Girls workshop!

Excited about learning Django, right?

So, without any delay let’s begin!

Have you ever wondered how to create a website but have never had enough motivation to start? Have you ever thought that the software world is too complicated for you to even try doing something on your own?

Well, then we have a good news!! We, today are going to make a website of our own(yayy)!!

We hope that we'll be able to make you love technology as much as we do!

LET’S START!!!!!

**IMPORTANT**

* Installation of Python
* Setting up a Virtual Environment
* Why??
* Because, Virtualenv will isolate your Python/Django setup on a per-project basis. This means that any changes you make to one website won't affect any others you're also developing. Neat, right?
* Installing Django
* Installing a code editor (**Atom**, in our case!)
* Installing Git
* Making a GitHub account
* Making a Python Anywhere account

TIME TO BEGIN OUR TEHNOLOGICAL RIDE FOR TODAY!!

(We hope it never ends (wink)!!!)

**HOW DOES THE INTERNET WORK?**

Imagine that when you type [https://djangogirls.org](https://djangogirls.org/), you send a letter that says: "Dear Django Girls, I want to see the djangogirls.org website. Send it to me, please!"

Your letter goes to the post office closest to you. Then it goes to another that is a bit nearer to your addressee, then to another, and another until it is delivered at its destination. The only unique thing is that if you send many letters (data packets) to the same place, they could go through totally different post offices (routers). This depends on how they are distributed at each office.

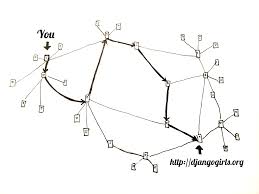
That's how it works - you send messages and you expect some response. Of course, instead of paper and pen you use bytes of data, but the idea is the same!

Instead of addresses with a street name, city, zip code and country name, we use IP addresses. Your computer first asks the DNS (Domain Name System) to translate djangogirls.org into an IP address. It works a little bit like old-fashioned phonebooks where you can look up the name of the person you want to contact and find their phone number and address.

When you send a letter, it needs to have certain features to be delivered correctly: an address, a stamp, etc. You also use a language that the receiver understands, right? The same applies to the data packets you send to see a website. We use a protocol called HTTP (Hypertext Transfer Protocol).

So, basically, when you have a website, you need to have a server (machine) where it lives. When the server receives an incoming request (in a letter), it sends back your website (in another letter).

A picture would help!



**INTRODUCTION TO COMMAND LINE**

To use command line (the standard black window!!) will surely give you the programmer feels, isn’t it?

The window, which is usually called the **command line** or **command-line interface**, is a text-based application for viewing, handling, and manipulating files on your computer.

Well, to start the command prompt,

Go to Start menu → Windows System → Command Prompt

Some important commands:

* cd – Current Directory
* whoami – Your Username
* dir – list of all files and directories
* cd <directory-name> - changes the directory to <directory-name>
* mkdir <directory-name> - creates directory of the <directory-name>

Now FINALLYYYYY!!!!

**PYTHON BASICS**

You can work with Python using your command lone as well as the command prompt that python provides you with (which is known as Idle, btw!)

 Let's start by typing some math, like 2 + 3 and hitting enter.

command-line

>>> 2 + 3

5

Practicing some more on our own...

STRINGS:

Well, Strings work well with python as well!

command-line

>>> "Ola"

'Ola'

>>> "Hi there " + "Ola"

'Hi there Ola'

>>> "Ola" \* 3

'OlaOlaOla'

>>> "Ola".upper()

'OLA'

>>> len("Ola")

3

VARIABLES:

>>> name = "Sonja"

>>> name

'Sonja'

>>> a = 4

>>> b = 6

>>> a \* b

24

>>> name = 'Maria'

>>> name

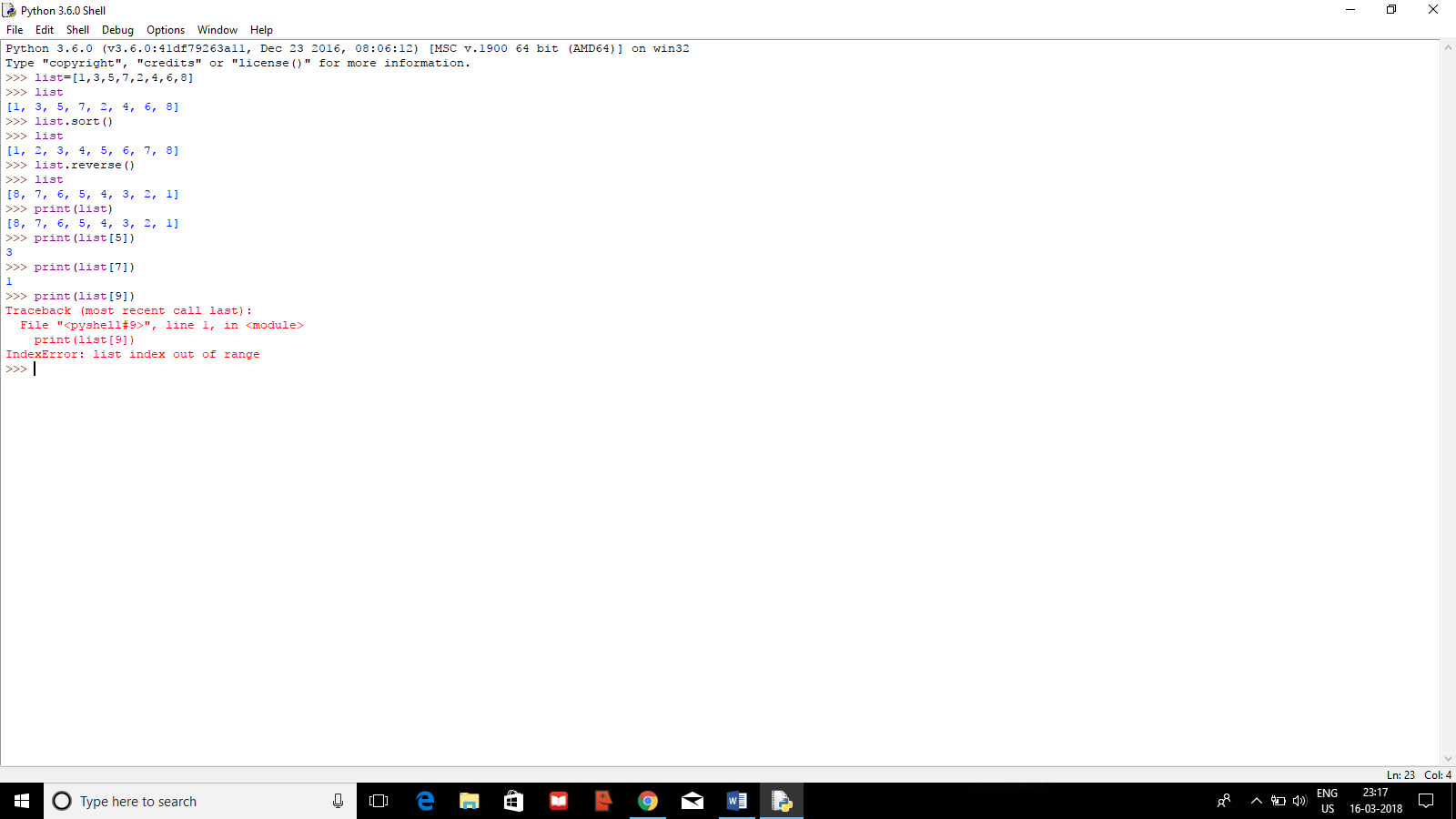
'Maria'

>>> print(name)

Maria

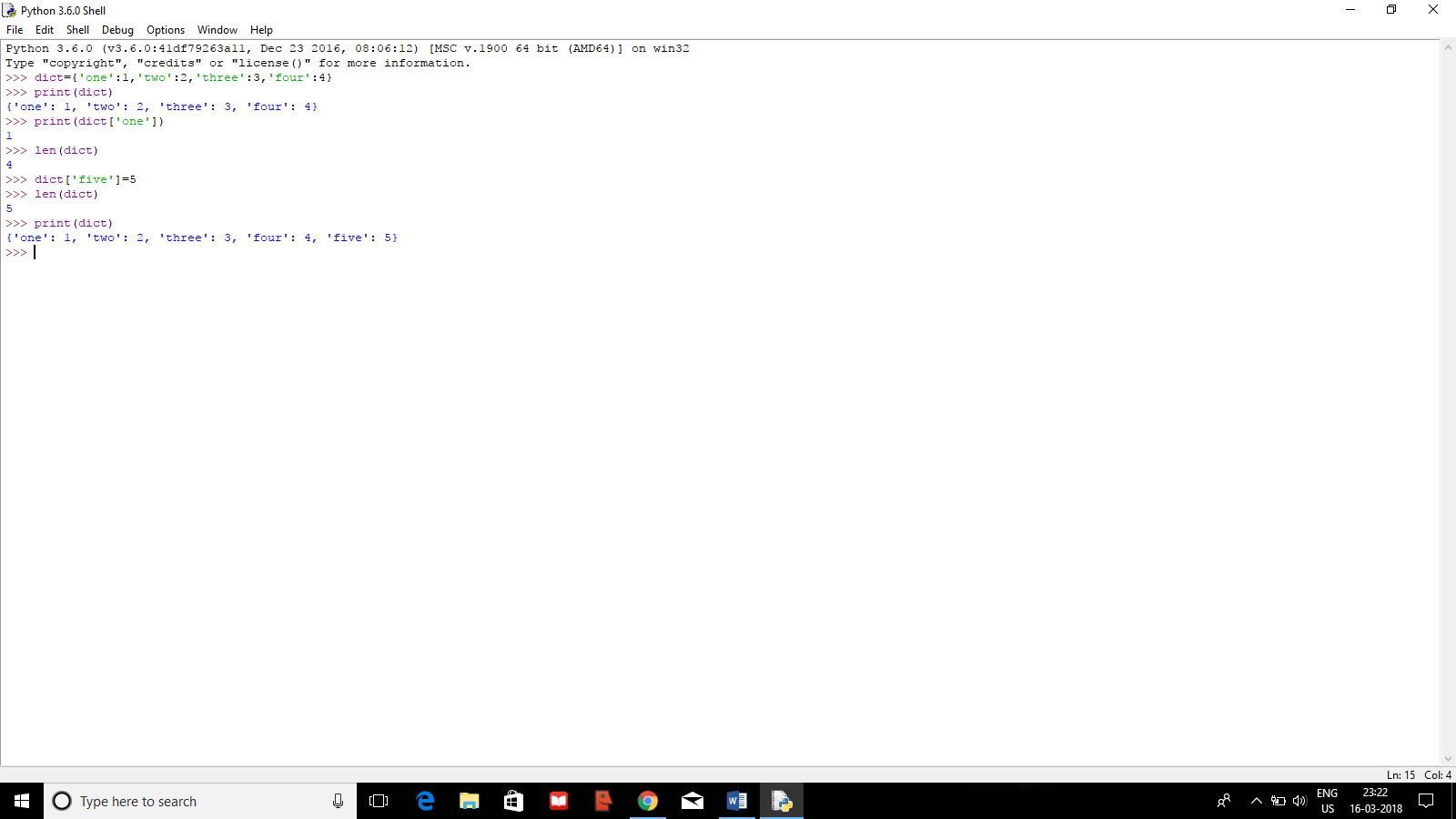
LISTS:

Lists are exactly what you think they are: objects which are lists of other objects.

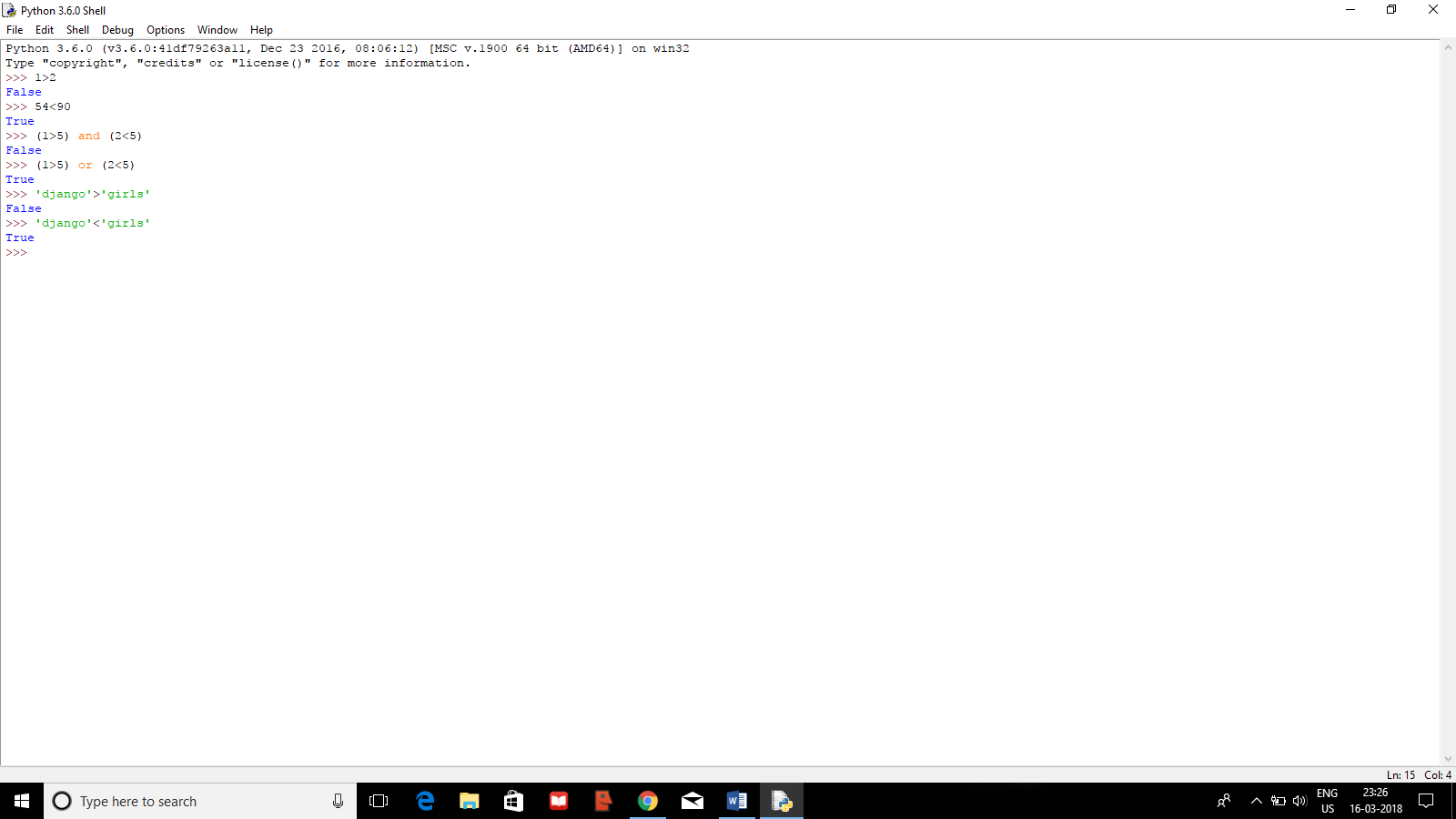


DICTIONARIES:

A dictionary is similar to a list, but you access values by looking up a key instead of a numeric index. A key can be any string or number.



COMPARISONS AND BOOLEANS:



If else STATEMENTS:

num=int(input("Enter a number:"))

if num > 10:

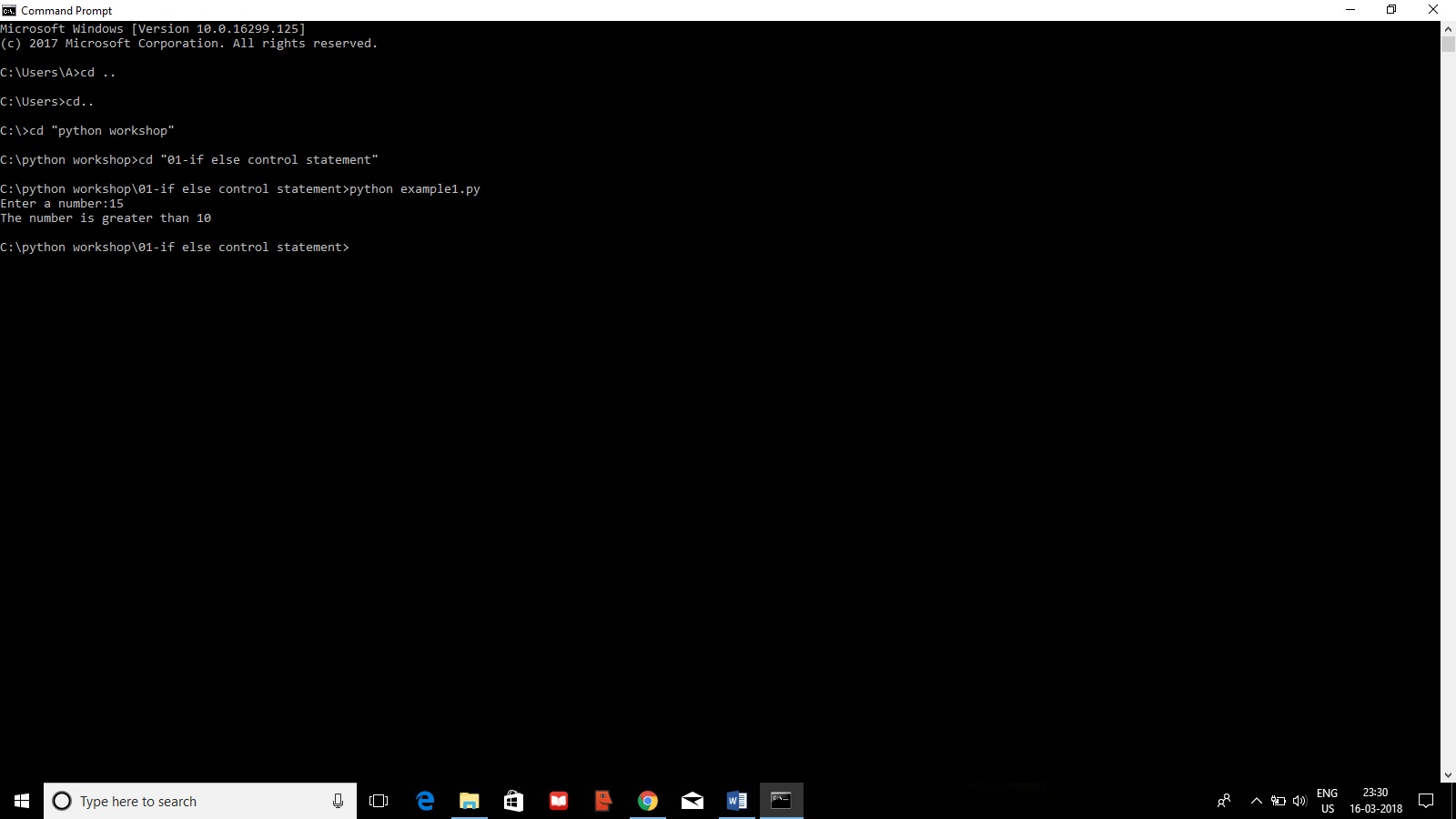
print("The number is greater than 10")

elif num < 10:

print('The number is lesser than 10')

else:

print("It seems you inputted 10")



FUNCTIONS:

def maximum(a,b):

if a>b:

print("{} is greater than {}".format(a,b))

elif a<b:

print("{} is greater than {}".format(b,a))

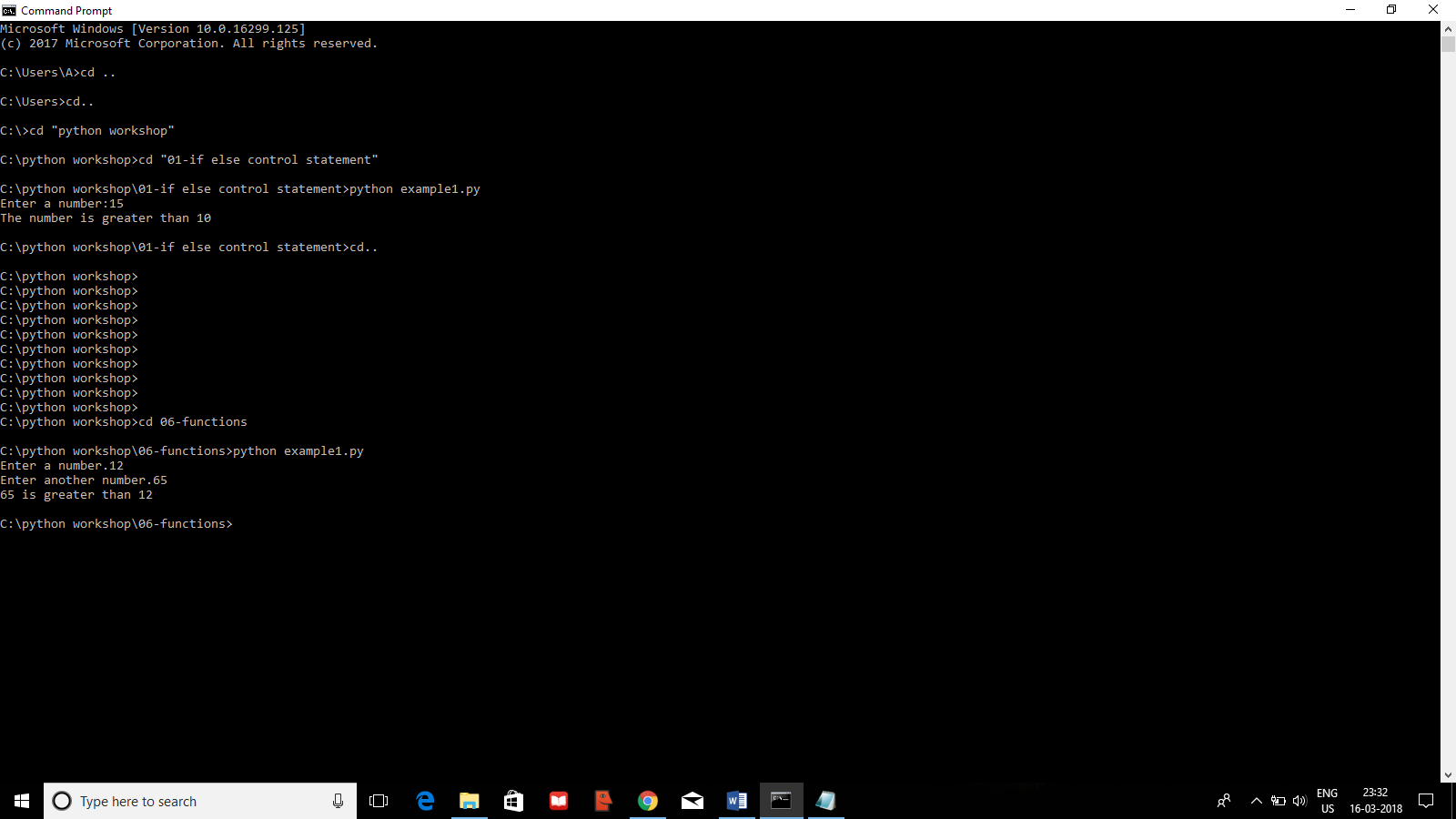
else:

print("{}={}".format(a,b))

a=int(input("Enter a number."))

b=int(input("Enter another number."))

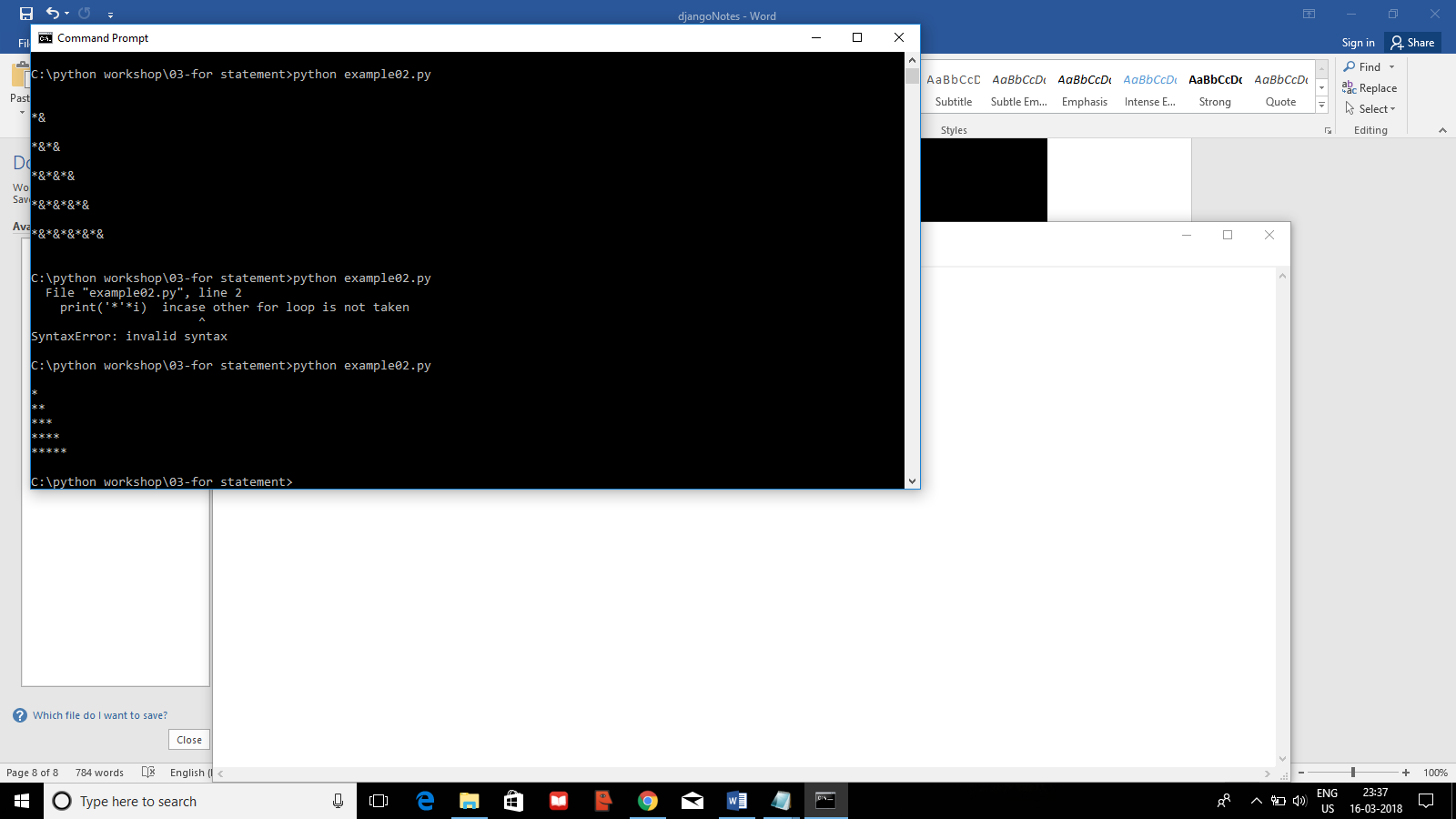
maximum(a,b)



LOOPS:

for i in range(6):

print('\*'\*i)



**What is Django?**

Django is a free and open source web application framework, written in Python. A web framework is a set of components that helps you to develop websites faster and easier.

When you're building a website, you always need a similar set of components: a way to handle user authentication (signing up, signing in, signing out), a management panel for your website, forms, a way to upload files, etc.

Luckily for you, other people long ago noticed that web developers face similar problems when building a new site, so they teamed up and created frameworks (Django being one of them) that give you ready-made components to use.

Frameworks exist to save you from having to reinvent the wheel and to help alleviate some of the overhead when you’re building a new site.

## **What happens when someone requests a website from your server?**

When a request comes to a web server, it's passed to Django which tries to figure out what is actually requested. It takes a web page address first and tries to figure out what to do. This part is done by Django's **urlresolver** (note that a website address is called a URL – Uniform Resource Locator – so the name urlresolver makes sense). It is not very smart – it takes a list of patterns and tries to match the URL. Django checks patterns from top to bottom and if something is matched, then Django passes the request to the associated function (which is called view).

**CREATING A PROJECT**

(myvenv) C:\Users\Name\djangogirls> django-admin.exe startproject mysite .

SETTING UP DATABASE:

(myvenv) ~/djangogirls$ python manage.py migrate

STARTING THE WEB SERVER:

(myvenv) ~/djangogirls$ python manage.py runserver

**DJANGO MODELS**

A model in Django is a special kind of object – it is saved in the database. A database is a collection of data. This is a place in which you will store information about users, your blog posts, etc. We will be using a SQLite database to store our data. This is the default Django database adapter – it'll be enough for us right now.

CREATING AN APP:

(myvenv) C:\Users\Name\djangogirls> python manage.py startapp blog

To register your app in the website, you must add it in the settings.py file in the list of the INSTALLED\_APPS.

blog/models.py:

from django.db import models

from django.utils import timezone

class Post(models.Model):

author = models.ForeignKey('auth.User', on\_delete=models.CASCADE)

title = models.CharField(max\_length=200)

text = models.TextField()

created\_date = models.DateTimeField(

default=timezone.now)

published\_date = models.DateTimeField(

blank=True, null=True)

def publish(self):

self.published\_date = timezone.now()

self.save()

def \_\_str\_\_(self):

return self.title

That is super-lengthy!!!!

We’ll see what each of them means….

class Post(models.Model): – this line defines our model (it is an object).

* class is a special keyword that indicates that we are defining an object.
* Post is the name of our model. We can give it a different name (but we must avoid special characters and whitespace). Always start a class name with an uppercase letter.
* models.Model means that the Post is a Django Model, so Django knows that it should be saved in the database.
* models.CharField – this is how you define text with a limited number of characters.
* models.TextField – this is for long text without a limit. Sounds ideal for blog post content, right?
* models.DateTimeField – this is a date and time.
* models.ForeignKey – this is a link to another model.

### Create tables for models in your database

The last step here is to add our new model to our database. First we have to make Django know that we have some changes in our model. (We have just created it!) Go to your console window and type python manage.py makemigrations blog. It will look like this:

command-line

(myvenv) ~/djangogirls$ python manage.py makemigrations blog

(myvenv) ~/djangogirls$ python manage.py migrate blog

# Django admin

To add, edit and delete the posts we've just modelled, we will use Django admin.

Let's open the blog/admin.py file and replace its contents with this:

blog/admin.py

from django.contrib import admin

from .models import Post

admin.site.register(Post)

But, who will be given the admin rights??  
Wondered??

Well, you need to create a super user for that who will act as admin.

(myvenv) C:\Users\Name\djangogirls> python manage.py createsuperuser

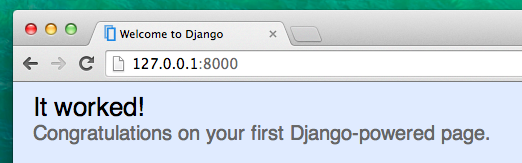
**TIME FOR urls**

# Django URLs

We're about to build our first webpage: a homepage for your blog! But first, let's learn a little bit about Django URLs.

## What is a URL?

A URL is simply a web address. You can see a URL every time you visit a website – it is visible in your browser's address bar. (Yes! 127.0.0.1:8000 is a URL! And https://djangogirls.org is also a URL.)



Every page on the Internet needs its own URL. This way your application knows what it should show to a user who opens that URL. In Django, we use something called URLconf (URL configuration). URLconf is a set of patterns that Django will try to match the requested URL to find the correct view.

## How do URLs work in Django?

Let's open up the mysite/urls.py file in your code editor of choice and see what it looks like:

mysite/urls.py

"""mysite URL Configuration

[...]

"""

from django.conf.urls import url

from django.contrib import admin

urlpatterns = [

url(r'^admin/', admin.site.urls),

]

As you can see, Django has already put something here for us.

Lines between triple quotes (''' or """) are called docstrings – you can write them at the top of a file, class or method to describe what it does. They won't be run by Python.

The admin URL, which you visited in the previous chapter, is already here:

mysite/urls.py

url(r'^admin/', admin.site.urls),

This line means that for every URL that starts with admin/, Django will find a corresponding view. In this case, we're including a lot of admin URLs so it isn't all packed into this small file – it's more readable and cleaner.

## Regex

Do you wonder how Django matches URLs to views? Well, this part is tricky. Django uses regex, short for "regular expressions". Regex has a lot (a lot!) of rules that form a search pattern. Since regexes are an advanced topic, we will not go into detail over how they work.

If you still wish to understand how we created the patterns, here is an example of the process – we will only need a limited subset of the rules to express the pattern we are looking for, namely:

* ^ for the beginning of the text
* $ for the end of the text
* \d for a digit
* + to indicate that the previous item should be repeated at least once
* () to capture part of the pattern

Anything else in the URL definition will be taken literally.

Now imagine you have a website with the address like http://www.mysite.com/post/12345/, where 12345 is the number of your post.

Writing separate views for all the post numbers would be really annoying. With regular expressions, we can create a pattern that will match the URL and extract the number for us: ^post/(\d+)/$. Let's break this down piece by piece to see what we are doing here:

* **^post/** is telling Django to take anything that has post/ at the beginning of the URL (right after ^)
* **(\d+)** means that there will be a number (one or more digits) and that we want the number captured and extracted
* **/** tells Django that another / character should follow
* **$** then indicates the end of the URL meaning that only strings ending with the / will match this pattern

## Your first Django URL!

Time to create our first URL! We want '<http://127.0.0.1:8000/>' to be the home page of our blog and to display a list of posts.

We also want to keep the mysite/urls.py file clean, so we will import URLs from our blog application to the main mysite/urls.py file.

Go ahead, add a line that will import blog.urls. Note that we are using the include function here so you will need to add that import.

Your mysite/urls.py file should now look like this:

mysite/urls.py

from django.conf.urls import include

from django.conf.urls import url

from django.contrib import admin

urlpatterns = [

url(r'^admin/', admin.site.urls),

url(r'', include('blog.urls')),

]

Django will now redirect everything that comes into '<http://127.0.0.1:8000/>' to blog.urls and looks for further instructions there.

Writing regular expressions in Python is always done with r in front of the string. This is a helpful hint for Python that the string may contain special characters that are not meant for Python itself, but for the regular expression instead.

## blog.urls

Create a new empty file named urls.py in the blog directory. All right! Add these first two lines:

blog/urls.py

from django.conf.urls import url

from . import views

Here we're importing Django's function url and all of our views from the blog application. (We don't have any yet, but we will get to that in a minute!)

After that, we can add our first URL pattern:

blog/urls.py

urlpatterns = [

url(r'^$', views.post\_list, name='post\_list'),

]

But, what is post\_list?? Let’s see!!!

**TIME TO CREATE VIEWS**

def post\_list(request):

return render(request, 'blog/post\_list.html', {})

As you can see, we created a function (def) called post\_list that takes request and return a function render that will render (put together) our template blog/post\_list.html.

But where is post\_list.html???

Well, we need to create it now!!!!

BUT, WHAT IS HTML???

HTML is a code that is interpreted by your web browser – such as Chrome, Firefox or Safari – to display a web page for the user.

HTML stands for "HyperText Markup Language". **HyperText** means it's a type of text that supports hyperlinks between pages. **Markup** means we have taken a document and marked it up with code to tell something (in this case, a browser) how to interpret the page. HTML code is built with **tags**, each one starting with < and ending with >. These tags represent markup **elements**.

Templates are saved in blog/templates/blog directory. So first create a directory called templatesinside your blog directory. Then create another directory called blog inside your templates directory

## **What is a QuerySet?**

A QuerySet is, in essence, a list of objects of a given Model. QuerySets allow you to read the data from the database, filter it and order it.

Fire the command “python manage.py shell”…

You will get a console with >>>, which is Python shell..

GO ahead and type…..

>>> from blog.models import Post

We import the model Post from blog.models. Let's try displaying all posts again:

command-line

>>> Post.objects.all()

### Create object

This is how you create a new Post object in database:

command-line

>>> Post.objects.create(author=me, title='Sample title', text='Test')

But we have one missing ingredient here: me. We need to pass an instance of User model as an author. How do we do that?

Let's import User model first:

command-line

>>> from django.contrib.auth.models import User

What users do we have in our database? Try this:

command-line

>>> User.objects.all()

This is the superuser we created earlier! Let's get an instance of the user now:

command-line

>>> me = User.objects.get(username='admin')

Now, you can create the Post object using “author=me”.

You can go ahead and create more objects…

Now, is the time to filter them….

>>> Post.objects.filter(author=me)

Got the expected output??

Ordering them….

>>> Post.objects.order\_by('created\_date')

You can chain them into one…

>>> Post.objects.filter(published\_date\_\_lte=timezone.now()).order\_by('published\_date')

You can even publish any post you want…

>>>post=Post.objects.get(title=’1st blog’)

And then go ahead and write – post.publish()

**DYNAMIC DATA IN TEMPLATES**

Well, in order to use the data of the posts created we need to make some changes in the views file as well as the post\_list.html file.

What changes??

Let’s see!!

blog/views.py

from django.shortcuts import render

from django.utils import timezone

from .models import Post

def post\_list(request):

posts = Post.objects.filter(published\_date\_\_lte=timezone.now()).order\_by('published\_date')

return render(request, 'blog/post\_list.html', {'posts': posts})

DJANGO TEMPLATES:

Now, go on and try typing {{posts}} in your post\_list.html file.

Saw the output??  
Which means it is understanding the query set as list of objects.

Let’s make it more readable and better.

blog/templates/blog/post\_list.html:

<div>

<h1><a href="/">Django Girls Blog</a></h1>

</div>

{% for post in posts %}

<div>

<p>published: {{ post.published\_date }}</p>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

Now, is the time for making our website more pretty and beautiful!!!

And answer to that is CSS(Cascading Style Sheets)!!

Well, learning everything about style sheets is not possible…

But, we could learn the basics, which would be just fine!!!

**What is CSS?**

Cascading Style Sheets (CSS) is a language used for describing the look and formatting of a website written in a markup language (like HTML). Treat it as make-up for our web page.

But we don't want to start from scratch again, right? Once more, we'll use something that programmers released on the Internet for free. Reinventing the wheel is no fun, you know.

**Let's use Bootstrap!**

Bootstrap is one of the most popular HTML and CSS frameworks for developing beautiful websites:

<https://getbootstrap.com/>

## Install Bootstrap

To install Bootstrap, you need to add this to your <head> in your .html file:

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/

css/bootstrap.min.css">

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/

css/bootstrap-theme.min.css">

## Static files in Django

Finally we will take a closer look at these things we've been calling **static files**. Static files are all your CSS and images. Their content doesn't depend on the request context and will be the same for every user.

Let's create a CSS file now, to add your own style to your web page. Create a new directory called css inside your static directory. Then create a new file called blog.css inside this css directory. Ready?

djangogirls

└─── blog

└─── static

└─── css

└─── blog.css

Time to write some CSS! Open up the blog/static/css/blog.css file in your code editor.

h1 a {

color: #FCA205;

}

h1 a is a CSS Selector. This means we're applying our styles to any a element inside of an h1element. So when we have something like <h1><a href="">link</a></h1>, the h1 a style will apply. In this case, we're telling it to change its color to #FCA205, which is orange. Of course, you can put your own color here!

We also need to tell our HTML template that we added some CSS.

Open the blog/templates/blog/post\_list.html file and add this line at the very beginning of it:

blog/templates/blog/post\_list.html

{% load staticfiles %}

We're just loading static files here. :) Between the <head> and </head> tags, after the links to the Bootstrap CSS files, add this line:

blog/templates/blog/post\_list.html

<link rel="stylesheet" href="{% static 'css/blog.css' %}">

The browser reads the files in the order they're given, so we need to make sure this is in the right place. Otherwise the code in our file may be overriden by code in Bootstrap files. We just told our template where our CSS file is located.

Nice work! Maybe we would also like to give our website a little air and increase the margin on the left side? Let's try this!

blog/static/css/blog.css

body {

padding-left: 15px;

}

we can customize the font in our header? Paste this into your <head> in blog/templates/blog/post\_list.html file:

blog/templates/blog/post\_list.html

<link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" rel="stylesheet" type="text/css">

Find the h1 a declaration block (the code between braces { and }) in the CSS file blog/static/css/blog.css. Now add the line font-family: 'Lobster'; between the braces, and refresh the page:

blog/static/css/blog.css

h1 a {

color: #FCA205;

font-family: 'Lobster';

}

Now refresh your page.

Go ahead and name some parts of the HTML code. Add a class called page-header to your div that contains your header, like this:

blog/templates/blog/post\_list.html

<div class="page-header">

<h1><a href="/">Django Girls Blog</a></h1>

</div>

And remaining body part is replace with this.

blog/templates/blog/post\_list.html

<div class="content container">

<div class="row">

<div class="col-md-8">

{% for post in posts %}

<div class="post">

<div class="date">

<p>published: {{ post.published\_date }}</p>

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

</div>

</div>

</div>

We will now add declaration blocks to different selectors. Selectors starting with . relate to classes.

blog/static/css/blog.css

.page-header {

background-color: #ff9400;

margin-top: 0;

padding: 20px 20px 20px 40px;

}

.page-header h1, .page-header h1 a, .page-header

h1 a:visited, .page-header h1 a:active {

color: #ffffff;

font-size: 36pt;

text-decoration: none;

}

.content {

margin-left: 40px;

}

h1, h2, h3, h4 {

font-family: 'Lobster', cursive;

}

.date {

color: #828282;

}

.save {

float: right;

}

.post-form textarea, .post-form input {

width: 100%;

}

.top-menu, .top-menu:hover, .top-menu:visited {

color: #ffffff;

float: right;

font-size: 26pt;

margin-right: 20px;

}

.post {

margin-bottom: 70px;

}

.post h1 a, .post h1 a:visited {

color: #000000;

}

Save those files and refresh your website.

# Template extending

It means that you can use the same parts of your HTML for different pages of your website.

Templates help when you want to use the same information or layout in more than one place. You don't have to repeat yourself in every file. And if you want to change something, you don't have to do it in every template, just one!

**Create Base Template :**

A base template is the most basic template that you extend on every page of your website.

Let's create a base.html file in blog/templates/blog/:

blog

└───templates

└───blog

base.html

post\_list.html

Then open it up and copy everything from post\_list.html to base.html file.

Then in base.html, replace your whole <body>

blog/templates/blog/base.html

<body>

<div class="page-header">

<h1><a href="/">Django Girls Blog</a></h1>

</div>

<div class="content container">

<div class="row">

<div class="col-md-8">

{% block content %}

{% endblock %}

</div>

</div>

</div>

</body>

You might notice this replaced everything from {% for post in posts %} to {% endfor %} with:

blog/templates/blog/base.html

{% block content %}

{% endblock %}

Now write below code snippet in the post\_list.html file

{% block content %}

{% for post in posts %}

<div class="post">

<div class="date">

{{ post.published\_date }}

</div>

<h1><a href="">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

{% endblock %}

And to connect these two templates together. This is what extending templates is all about! We'll do this by adding an extends tag to the beginning of the file. Like this:

blog/templates/blog/post\_list.html

{% extends 'blog/base.html' %}

Time to extend the application:

**EXTENDING THE APPLICATION:**

Let’s add a link in the blog/post\_list.html that will take us the details of a single post!

For that, the changes expected in the blog/post\_list.html are as follows:

You need to add a hyperlink to every post displayed on the post\_list.html page.

blog/post\_list.html:

{% extends 'blog/base.html' %}

{% block content %}

{% for post in posts %}

<div class="post">

<div class="date">

{{ post.published\_date }}

</div>

<h1><a href= "{% url 'post\_detail' pk=post.pk %}">{{ post.title }}</a></h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endfor %}

{% endblock %}

The post\_detail part means that Django will be expecting a URL in blog/urls.py with name=post\_detail  
And, we have not added any url with name=post\_detail.

And how about pk=post.pk? pk is short for primary key, which is a unique name for each record in a database. Because we didn't specify a primary key in our Post model, Django creates one for us (by default, a number that increases by one for each record, i.e. 1, 2, 3) and adds it as a field named pk to each of our posts. We access the primary key by writing post.pk, the same way we access other fields (title, author, etc.) in our Post object!

Let’s check if it works well…

Ohhhhh!!!!!!

There is an ERROR!!!

Why so???

Let's create a URL in urls.py for our post\_detail view!

from django.conf.urls import url

from . import views

urlpatterns = [

url(r'^$', views.post\_list, name='post\_list'),

url(r'^post/(?P<pk>\d+)/$', views.post\_detail, name='post\_detail'),

]

This part ^post/(?P<pk>\d+)/$ looks scary, but no worries – we will explain it for you:

* it starts with ^ again – "the beginning".
* post/ just means that after the beginning, the URL should contain the word **post** and a **/**. So far so good.
* (?P<pk>\d+) – this part is trickier. It means that Django will take everything that you place here and transfer it to a view as a variable called pk. (Note that this matches the name we gave the primary key variable back in blog/templates/blog/post\_list.html!) \d also tells us that it can only be a digit, not a letter (so everything between 0 and 9). + means that there needs to be one or more digits there. So something like http://127.0.0.1:8000/post// is not valid, but http://127.0.0.1:8000/post/1234567890/ is perfectly OK!
* / – then we need a **/** again.
* $ – "the end"!

And the next step is adding a view!!

Go to blog/views.py and add a new view:

blog/views.py:

def post\_detail(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

return render(request, 'blog/post\_detail.html', {'post': post})

What is get\_object\_or\_404??

The good news is that you can actually create your own Page not found page and make it as pretty as you want. But it's not super important right now, so we will skip it.

## Create a template for the post details

We will create a file in blog/templates/blog called post\_detail.html.

It will look like this:

blog/templates/blog/post\_detail.html

{% extends 'blog/base.html' %}

{% block content %}

<div class="post">

{% if post.published\_date %}

<div class="date">

{{ post.published\_date }}

</div>

{% endif %}

<h1>{{ post.title }}</h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endblock %}

Refresh!!!!

Yayyy!!!! It works!!!

# Django Forms

The final thing we want to do on our website is create a nice way to add and edit blog posts. Django's admin is cool, but it is rather hard to customize and make pretty. With forms we will have absolute power over our interface – we can do almost anything we can imagine!

The nice thing about Django forms is that we can either define one from scratch or create a ModelForm which will save the result of the form to the model.

This is exactly what we want to do: we will create a form for our Post model.

Like every important part of Django, forms have their own file: forms.py.

We need to create a file with this name in the blog directory.

blog

└── forms.py

blog/forms.py

from django import forms

from .models import Post

class PostForm(forms.ModelForm):

class Meta:

model = Post

fields = ('title', 'text',)

We need to import Django forms first (from django import forms) and, obviously, our Post model (from .models import Post).

PostForm, as you probably suspect, is the name of our form. We need to tell Django that this form is a ModelForm (so Django will do some magic for us) – forms.ModelForm is responsible for that.

Next, we have class Meta, where we tell Django which model should be used to create this form (model = Post).

Finally, we can say which field(s) should end up in our form. In this scenario we want only title and text to be exposed – author should be the person who is currently logged in (you!) and created\_date should be automatically set when we create a post (i.e. in the code), right?

And that's it! All we need to do now is use the form in a *view* and display it in a template.

So once again we will create a link to the page, a URL, a view and a template.

Note that we want to call our new view post\_new. The class "glyphicon glyphicon-plus" is provided by the bootstrap theme we are using, and will display a plus sign for us.

After adding the line, your HTML file should now look like this:

blog/templates/blog/base.html

{% load staticfiles %}

<html>

<head>

<title>Django Girls blog</title>

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

<link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'>

<link rel="stylesheet" href="{% static 'css/blog.css' %}">

</head>

<body>

<div class="page-header">

<a href="{% url 'post\_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>

<h1><a href="/">Django Girls Blog</a></h1>

</div>

<div class="content container">

<div class="row">

<div class="col-md-8">

{% block content %}

{% endblock %}

</div>

</div>

</div>

</body>

</html>

URL:

blog/urls.py

from django.conf.urls import url

from . import views

urlpatterns = [

url(r'^$', views.post\_list, name='post\_list'),

url(r'^post/(?P<pk>\d+)/$', views.post\_detail, name='post\_detail'),

url(r'^post/new/$', views.post\_new, name='post\_new'),

]

## post\_new view

Time to open the blog/views.py file and add the following lines with the rest of the from rows:

blog/views.py

from .forms import PostForm

And then our view:

blog/views.py

def post\_new(request):

form = PostForm()

return render(request, 'blog/post\_edit.html', {'form': form})

To create a new Post form, we need to call PostForm() and pass it to the template. We will go back to this view, but for now, let's quickly create a template for the form.

## Template

We need to create a file post\_edit.html in the blog/templates/blog directory. To make a form work we need several things:

* We have to display the form. We can do that with (for example) {{ form.as\_p }}.
* The line above needs to be wrapped with an HTML form tag: <form method="POST">...</form>.
* We need a Save button. We do that with an HTML button: <button type="submit">Save</button>.
* And finally, just after the opening <form ...> tag we need to add {% csrf\_token %}. This is very important, since it makes your forms secure! If you forget about this bit, Django will complain when you try to save the form:

But, an ERROR!!!!!

Let’s make some changes in post\_edit.html:

{% extends 'blog/base.html' %}

{% block content %}

<h1>New post</h1>

<form method="POST" class="post-form">{% csrf\_token %}

{{ form.as\_p }}

<button type="submit" class="save btn btn-default">Save</button>

</form>

{% endblock %}

Time to refresh! Yay! Your form is displayed!

But, wait a minute! When you type something in the title and text fields and try to save it, what will happen?

Nothing! We are once again on the same page and our text is gone… and no new post is added. So what went wrong?

The answer is: nothing. We need to do a little bit more work in our *view*.

## Saving the form

Open blog/views.py once again. Currently all we have in the post\_new view is the following:

blog/views.py

def post\_new(request):

form = PostForm()

return render(request, 'blog/post\_edit.html', {'form': form})

When we submit the form, we are brought back to the same view, but this time we have some more data in request, more specifically in request.POST (the naming has nothing to do with a blog "post"; it's to do with the fact that we're "posting" data). Remember how in the HTML file, our <form> definition had the variable method="POST"? All the fields from the form are now in request.POST. You should not rename POST to anything else (the only other valid value for method is GET, but we have no time to explain what the difference is).

So in our view we have two separate situations to handle: first, when we access the page for the first time and we want a blank form, and second, when we go back to the view with all form data we just typed. So we need to add a condition (we will use if for that):

blog/views.py

if request.method == "POST":

[...]

else:

form = PostForm()

It's time to fill in the dots [...]. If method is POST then we want to construct the PostForm with data from the form, right? We will do that as follows:

blog/views.py

form = PostForm(request.POST)

The next thing is to check if the form is correct (all required fields are set and no incorrect values have been submitted). We do that with form.is\_valid().

We check if the form is valid and if so, we can save it!

blog/views.py

if form.is\_valid():

post = form.save(commit=False)

post.author = request.user

post.published\_date = timezone.now()

post.save()

Basically, we have two things here: we save the form with form.save and we add an author (since there was no author field in the PostForm and this field is required). commit=False means that we don't want to save the Post model yet – we want to add the author first. Most of the time you will use form.save()without commit=False, but in this case, we need to supply it. post.save() will preserve changes (adding the author) and a new blog post is created!

Finally, it would be awesome if we could immediately go to the post\_detail page for our newly created blog post, right? To do that we need one more import:

blog/views.py

from django.shortcuts import redirect

Add it at the very beginning of your file. And now we can say, "go to the post\_detail page for the newly created post":

blog/views.py

return redirect('post\_detail', pk=post.pk)

post\_detail is the name of the view we want to go to. Remember that this view requires a pkvariable? To pass it to the views, we use pk=post.pk, where post is the newly created blog post!

OK, we've talked a lot, but we probably want to see what the whole view looks like now, right?

blog/views.py

def post\_new(request):

if request.method == "POST":

form = PostForm(request.POST)

if form.is\_valid():

post = form.save(commit=False)

post.author = request.user

post.published\_date = timezone.now()

post.save()

return redirect('post\_detail', pk=post.pk)

else:

form = PostForm()

return render(request, 'blog/post\_edit.html', {'form': form})

Let's see if it works. Go to the page <http://127.0.0.1:8000/post/new/>, add a title and text, save it… and EUREKA! The new blog post is added and we are redirected to the post\_detail page!

You might have noticed that we are setting the publish date before saving the post. Later on, we will introduce a publish button in **Django Girls Tutorial: Extensions**.

That is awesome!

## **Form validation**

Now, we will show you how cool Django forms are. A blog post needs to have title and text fields. In our Post model we did not say that these fields (as opposed to published\_date) are not required, so Django, by default, expects them to be set.

Try to save the form without title and text. Guess what will happen!

Ohh!!!!It works magic!!!

## **Edit form**

Now we know how to add a new form. But what if we want to edit an existing one? This is very similar to what we just did. Let's create some important things quickly. (If you don't understand something, you should ask your coach or look at the previous chapters, since we covered all these steps already.)

Open blog/templates/blog/post\_detail.html and add the line

blog/templates/blog/post\_detail.html

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

so that the template will look like this:

blog/templates/blog/post\_detail.html

{% extends 'blog/base.html' %}

{% block content %}

<div class="post">

{% if post.published\_date %}

<div class="date">

{{ post.published\_date }}

</div>

{% endif %}

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

<h1>{{ post.title }}</h1>

<p>{{ post.text|linebreaksbr }}</p>

</div>

{% endblock %}

In blog/urls.py we add this line:

blog/urls.py

url(r'^post/(?P<pk>\d+)/edit/$', views.post\_edit, name='post\_edit'),

We will reuse the template blog/templates/blog/post\_edit.html, so the last missing thing is a view.

Let's open blog/views.py and add this at the very end of the file:

blog/views.py

def post\_edit(request, pk):

post = get\_object\_or\_404(Post, pk=pk)

if request.method == "POST":

form = PostForm(request.POST, instance=post)

if form.is\_valid():

post = form.save(commit=False)

post.author = request.user

post.published\_date = timezone.now()

post.save()

return redirect('post\_detail', pk=post.pk)

else:

form = PostForm(instance=post)

return render(request, 'blog/post\_edit.html', {'form': form})

This looks almost exactly the same as our post\_new view, right? But not entirely. For one, we pass an extra pk parameter from urls. Next, we get the Post model we want to edit with get\_object\_or\_404(Post, pk=pk) and then, when we create a form, we pass this post as an instance, both when we save the form…

blog/views.py

form = PostForm(request.POST, instance=post)

…and when we've just opened a form with this post to edit:

blog/views.py

form = PostForm(instance=post)

OK, let's test if it works! Let's go to the post\_detail page. There should be an edit button in the top-right corner:

Does it work as expected???

Yes!!!

## Security

Being able to create new posts just by clicking a link is awesome! But right now, anyone who visits your site will be able to make a new blog post, and that's probably not something you want. Let's make it so the button shows up for you but not for anyone else.

In blog/templates/blog/base.html, find our page-header div and the anchor tag you put in there earlier. It should look like this:

blog/templates/blog/base.html

<a href="{% url 'post\_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>

We're going to add another {% if %} tag to this, which will make the link show up only for users who are logged into the admin. Right now, that's just you! Change the <a> tag to look like this:

blog/templates/blog/base.html

{% if user.is\_authenticated %}

<a href="{% url 'post\_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>

{% endif %}

This {% if %} will cause the link to be sent to the browser only if the user requesting the page is logged in. This doesn't protect the creation of new posts completely, but it's a good first step. We'll cover more security in the extension lessons.

Remember the edit icon we just added to our detail page? We also want to add the same change there, so other people won't be able to edit existing posts.

Open blog/templates/blog/post\_detail.html and find this line:

blog/templates/blog/post\_detail.html

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

Change it to this:

blog/templates/blog/post\_detail.html

{% if user.is\_authenticated %}

<a class="btn btn-default" href="{% url 'post\_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>

{% endif %}

And that’s it!!!!!

CONGRATULATIONS!!!!

You have successfully created your first website!!!