

## Python History

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by Guido Van Rossum at CWI in Netherland.
- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Python is influenced by following programming languages:
  - ABC language.
  - Modula-3

## Strengths and Weaknesses

### Strengths of Python

#### Extensive Support Libraries

It provides large standard libraries that include the areas like string operations, Internet, web service tools, operating system interfaces and protocols. Most of the highly used programming tasks are already scripted into it that limits the length of the codes to be written in Python.

#### Integration Feature

Python integrates the Enterprise Application Integration that makes it easy to develop Web services by invoking COM or COBRA components. It has powerful control capabilities as it calls directly through C, C++ or Java via Jython. Python also processes XML and other markup languages as it can run on all modern operating systems through same byte code.

#### Improved Programmer's Productivity

The language has extensive support libraries and clean object-oriented designs that increase two to ten fold of programmer's productivity while using the languages like Java, VB, Perl, C, C++ and C#.

#### Productivity

With its strong process integration features, unit testing framework and enhanced control capabilities contribute towards the increased speed for most applications and productivity of applications. It is a great option for building scalable multi-protocol network applications.

## Weakness of Python

### Difficulty in Using Other Languages

The Python lovers become so accustomed to its features and its extensive libraries, so they face problem in learning or working on other programming languages. Python experts may see the declaring of cast “values” or variable “types”, syntactic requirements of adding curly braces or semi colons as an onerous task.

### Weak in Mobile Computing

Python has made its presence on many desktop and server platforms, but it is seen as a weak language for mobile computing. This is the reason very few mobile applications are built in it like Carbonnelle.

### Gets Slow in Speed

Python executes with the help of an interpreter instead of the compiler, which causes it to slow down because compilation and execution help it to work normally. On the other hand, it can be seen that it is fast for many web applications too.

### Run-time Errors

The Python language is dynamically typed so it has many design restrictions that are reported by some Python developers. It is even seen that it requires more testing time, and the errors show up when the applications are finally run.

### Underdeveloped Database Access Layers

As compared to the popular technologies like JDBC and ODBC, the Python’s database access layer is found to be bit underdeveloped and primitive. However, it cannot be applied in the enterprises that need smooth interaction of complex legacy data.

## Python Versions.

Version	Released Date	Version	Released Date
Python 1.0	January 1994	Python 3.0	December 3, 2008
Python 1.5	December 31, 1997	Python 3.1	June 27, 2009
Python 1.6	September 5, 2000	Python 3.2	February 20, 2011
Python 2.0	October 16, 2000	Python 3.3	September 29, 2012
Python 2.1	April 17, 2001	Python 3.4	March 16, 2014
Python 2.2	December 21, 2001	Python 3.5	September 13, 2015
Python 2.3	July 29, 2003	Python 3.6	December 23, 2016
Python 2.4	November 30, 2004	Python 3.7	June 27, 2018
Python 2.5	September 19, 2006		
Python 2.6	October 1, 2008		

## Installing Python and setting up Environment Variables

It is highly unlikely that your Windows system shipped with Python already installed. Windows systems typically do not. Fortunately, installing does not involve much more than downloading the Python installer from the python.org website and running it. Let's take a look at how to install Python 3 on Windows:

### Step 1: Download the Python 3 Installer

1. Open a browser window and navigate to the Download page for Windows at python.org.
2. Underneath the heading at the top that says Python Releases for Windows, click on the link for the Latest Python 3 Release - Python 3.x.x.
3. Scroll to the bottom and select either Windows x86-64 executable installer for 64-bit or Windows x86 executable installer for 32-bit.

### Step 2: Run the Installer

Once you have chosen and downloaded an installer, simply run it by double-clicking on the downloaded file. A dialog should appear that looks something like this:

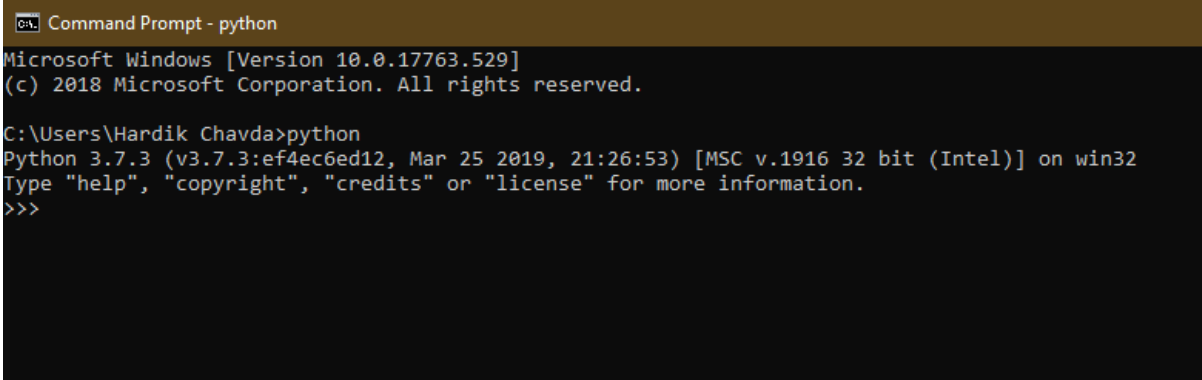


**Step 3:** You want to be sure to check the box that says **Add Python 3.x to PATH** as shown to ensure that the interpreter will be placed in your execution path.

**Step 4:** Then just click **Install Now**. That should be all there is to it. A few minutes later you should have a working Python 3 installation on your system.

## Executing Python from the Command Line

1. Open Command line: Start menu -> Run and type cmd
2. In cmd prompt type python
3. Python prompt will appear



```
Command Prompt - python
Microsoft Windows [Version 10.0.17763.529]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Hardik Chavda>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

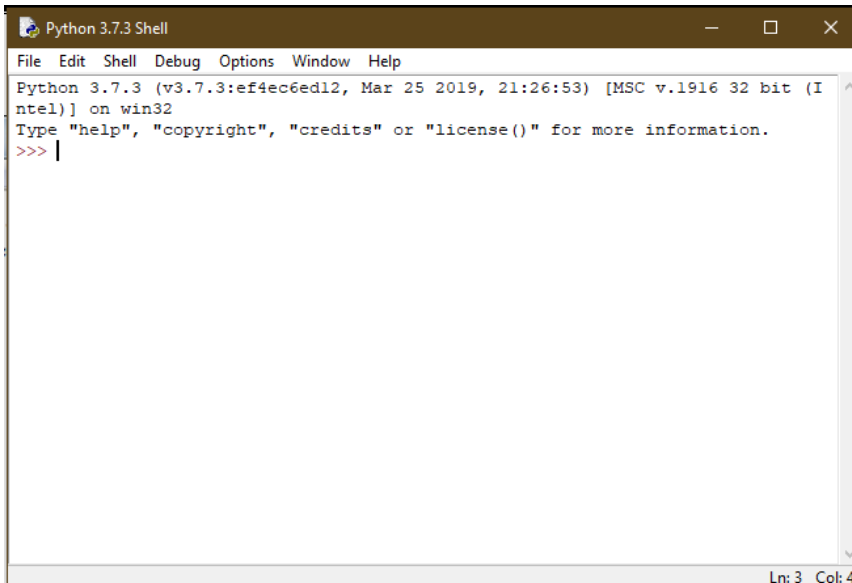
## IDLE

### Python - IDLE

IDLE (Integrated Development and Learning Environment) is an integrated development environment (IDE) for Python. The Python installer for Windows contains the IDLE module by default.

IDLE can be used to execute a single statement just like Python Shell and also to create, modify and execute Python scripts. IDLE provides a fully-featured text editor to create Python scripts that includes features like syntax highlighting, auto completion and smart indent. It also has a debugger with stepping and breakpoints features.

To start IDLE interactive shell, search for the IDLE icon in the start menu and double click on it.

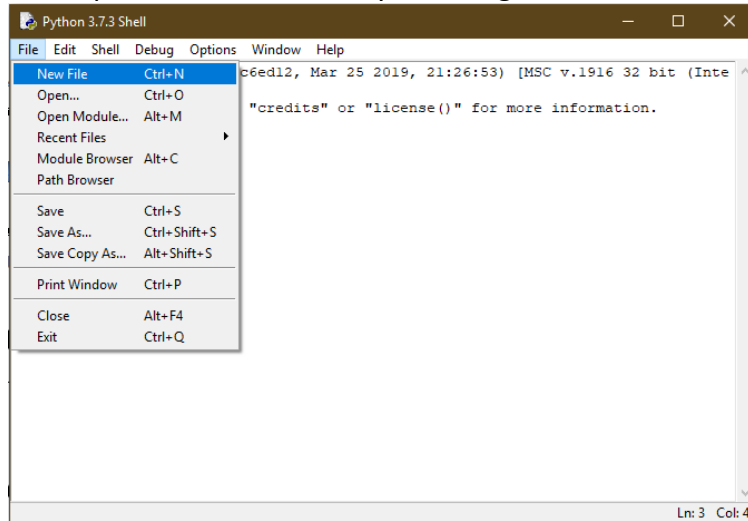


```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

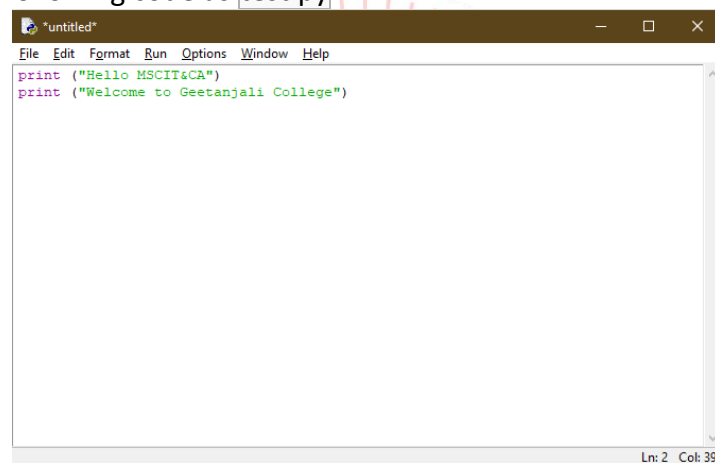
This will open IDLE, where you can write Python code and execute it as shown below.

## Editing Python Files

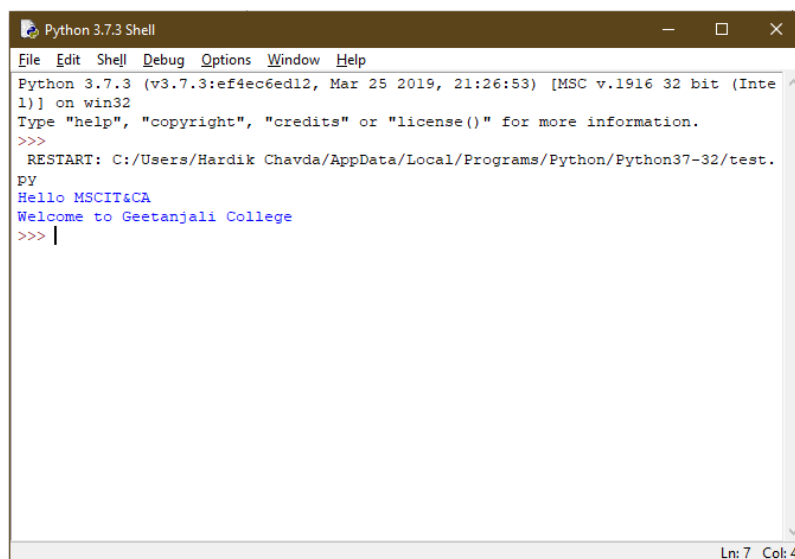
To execute a Python script, create a new file by selecting File -> New File from the menu.



Enter multiple statements and save the file with extension .py using File -> Save. For example, save the following code as test.py.



Now, press F5 to run the script in the editor window. The IDLE shell will show the output.

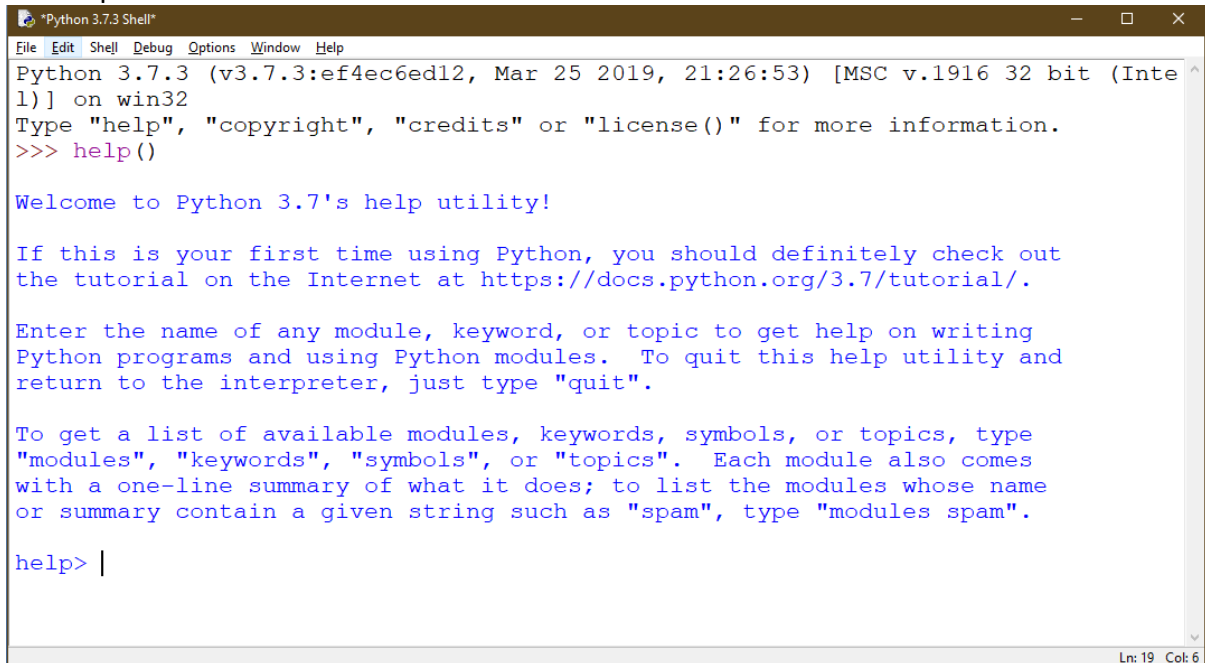


## Getting Help

The python help function is used to display the documentation of modules, functions, classes, keywords etc. The help function has the following syntax:

```
help([object])
```

If the help function is passed without an argument, then the interactive help utility starts up on the console.

A screenshot of a Python 3.7.3 Shell window. The title bar reads "Python 3.7.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following output:

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> help()

Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", type "modules spam".

help> |
```

The status bar at the bottom right indicates "Ln: 19 Col: 6".

The help() method is used for interactive use. It's recommended to try it in your interpreter when you need help to write Python program and use Python modules.

object is passed to help() (not a string)

Try these on Python shell.

```
>>> help(list)
>>> help(dict)
>>> help(print)
>>> help([1, 2, 3])
```

If string is passed as an argument, name of a module, function, class, method, keyword, or documentation topic, and a help page is printed.

## Dynamic Types

Python variable assignment is different from some of the popular languages like c, c++ and java. There is no declaration of a variable, just an assignment statement.

When we declare a variable in C or alike languages, this sets aside an area of memory for holding values allowed by the data type of the variable. The memory allocated will be interpreted as the data type suggests. If it's an integer variable the memory allocated will be read as an integer and so on. When we assign or initialize it with some value, that value will get stored at that memory location. At compile time, initial value or assigned value will be checked. So we cannot mix types. Example: initializing a string value to an int variable is not allowed and the program will not compile.

But Python is a dynamically typed language. It doesn't know about the type of the variable until the code is run. So declaration is of no use. What it does is, It stores that value at some memory location and then binds that variable name to that memory container. And makes the contents of the container accessible through that variable name. So the data type does not matter. As it will get to know the type of the value at run-time.

### Example

```
x = 6
print(type(x))
x = 'hello'
print(type(x))
```

### Output:

```
<class 'int'>
<class 'str'>
```

## Python Reserved Words

Keywords are the reserved words in Python. We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive. There are 33 keywords in Python 3.7. This number can vary slightly in the course of time. All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords is given below.

<b>False</b>	<b>class</b>	<b>finally</b>	<b>is</b>	<b>return</b>
<b>None</b>	<b>continue</b>	<b>for</b>	<b>lambda</b>	<b>try</b>
<b>True</b>	<b>def</b>	<b>from</b>	<b>nonlocal</b>	<b>while</b>
<b>and</b>	<b>del</b>	<b>global</b>	<b>not</b>	<b>with</b>
<b>as</b>	<b>elif</b>	<b>if</b>	<b>or</b>	<b>yield</b>
<b>assert</b>	<b>else</b>	<b>import</b>	<b>pass</b>	
<b>break</b>	<b>except</b>	<b>in</b>	<b>raise</b>	



## Naming Conventions

### 1. General

- Avoid using names that are too general or too wordy. Strike a good balance between the two.
- Bad: `data_structure`, `my_list`, `info_map`, `dictionary_for_the_purpose_of_storing_data_representing_word_definitions`
- When using CamelCase names, capitalize all letters of an abbreviation (e.g. `HTTPServer`)

### 2. Packages

- Package names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

### 3. Modules

- Module names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

### 4. Classes

- Class names should follow the UpperCaseCamelCase convention
- Python's built-in classes, however are typically lowercase words
- Exception classes should end in "Error"

### 5. Global (module-level) Variables

- Global variables should be all lowercase
- Words in a global variable name should be separated by an underscore

### 6. Instance Variables

- Instance variable names should be all lower case
- Words in an instance variable name should be separated by an underscore
- Non-public instance variables should begin with a single underscore
- If an instance name needs to be mangled, two underscores may begin its name

### 7. Methods

- Method names should be all lower case
- Words in a method name should be separated by an underscore
- Non-public method should begin with a single underscore
- If a method name needs to be mangled, two underscores may begin its name

### 8. Method Arguments

- Instance methods should have their first argument named 'self'.
- Class methods should have their first argument named 'cls'

### 9. Functions

- Function names should be all lower case
- Words in a function name should be separated by an underscore

### 10. Constants

- Constant names must be fully capitalized
- Words in a constant name should be separated by an underscore



## Basic Syntax

By default, the Python interpreter treats a piece of text terminated by hard carriage return (new line character) as one statement. It means each line in a Python script is a statement. (Just as in C/C++/C#, a semicolon ; denotes the end of a statement).

### Example:

```
msg="Hello World"  
code=123  
name="HardikChavda"
```

However, you can show the text spread over more than one lines to be a single statement by using the backslash (\) as a continuation character. Look at the following examples:

### Example: Continuation of Statement

```
msg="Hello Pythonista \  
Welcome to Python Tutorial \  
from GeetanjaliCollege"  
Similarly, use the semicolon ; to write multiple statements in a single line.  
Example: Multiple Statements in Single Line  
msg="Hello World";code=123;name=" HardikChavda "
```

## Indents in Python

Many times it is required to construct a block of more than one statements. For example there are usually multiple statements that are part of the definition of a function. There can be one or more statements in a looping construct.

Different programming languages use different techniques to define the scope and extent of a block of statements in constructs like class, function, conditional and loop. In C, C++, C# or Java, statements inside curly brackets { and } are treated as a block.

Python uses uniform indentation to denote a block of statements. When a block is to be started, type the exclamation symbol (:) and press Enter. Any Python-aware editor (like IDLE) goes to the next line leaving an additional whitespace (called indent). Subsequent statements in the block follow the same level of indent. In order to signal the end of a block, the whitespace is de-dented by pressing the backspace key. If your editor is not configured for Python, you may have to ensure that the statements in a block have the same indentation level by pressing the spacebar or Tab key. The Python interpreter will throw an error if the indentation level in the block is not same.

## Comments

In a Python script, the symbol # indicates the start of a comment line. It is effective till the end of the line in the editor. If # is the first character of the line, then the entire line is a comment. It can be used also in the middle of a line. The text before it is a valid Python expression, while the text following is treated as a comment.

### Example:

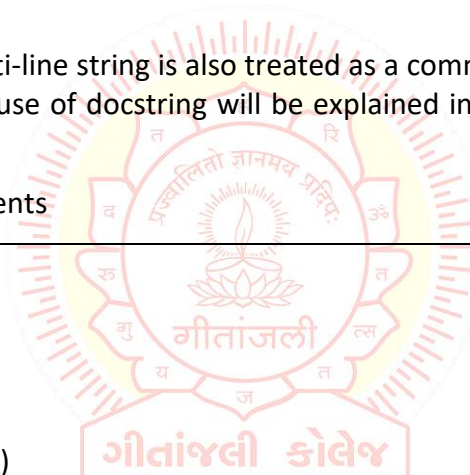
```
# this is a comment
print ("Hello World")
print ("Welcome to Python Tutorial") #this is comment but after a statement.
```

In Python, there is no provision to write multi-line comments, or a block comment. (As in C#/C/C++, where multiple lines inside /\* .. \*/ are treated as a multi-line comment). Each line should have the # symbol at the start to be marked as a comment. Many Python IDEs have shortcuts to mark a block of statements as a comment. In IDLE, select the block and press Alt + 3.

A triple quoted multi-line string is also treated as a comment if it is not a docstring of a function or a class. (The use of docstring will be explained in subsequent tutorials on Python functions.)

### Example: Multi-line Comments

```
"""
comment1
comment2
comment3
"""
print ("Hello World")
```



## String Values

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings

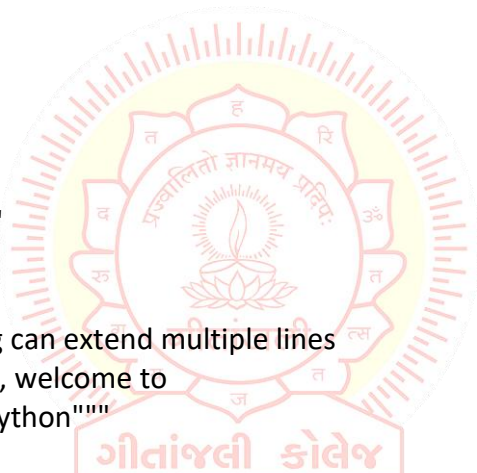
### Example

```
# all of the following are equivalent
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)

my_string = """Hello"""
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
the world of Python"""
print(my_string)
```



### Output

```
Hello
Hello
Hello
Hello, welcome to
the world of Python
```

## String Operations

### Multiline Strings

You can assign a multiline string to a variable by using three quotes:

You can use three double quotes:

#### Example

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
```

Or three single quotes:

#### Example

```
a = 'Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'
```

In python, strings are treated as the sequence of strings which means that python doesn't support the character data type instead a single character written as 'p' is treated as the string of length 1.

Python provides various in-built methods that are used for string handling. Many String fun

Method	Description
<b>capitalize()</b>	It capitalizes the first character of the String. This function is deprecated in python3
<b>casefold()</b>	It returns a version of s suitable for case-less comparisons.
<b>center(width ,fillchar)</b>	It returns a space padded string with the original string centred with equal number of left and right spaces.
<b>count(string,begin,end)</b>	It counts the number of occurrences of a substring in a String between begin and end index.
<b>decode(encoding = 'UTF8', errors = 'strict')</b>	Decodes the string using codec registered for encoding.
<b>encode()</b>	Encode S using the codec registered for encoding. Default encoding is 'utf-8'.
<b>endswith(suffix ,begin=0,end=len(string))</b>	It returns a Boolean value if the string terminates with given suffix between begin and end.
<b>expandtabs(tabsize = 8)</b>	It defines tabs in string to multiple spaces. The default space value is 8.
<b>find(substring ,beginIndex, endIndex)</b>	It returns the index value of the string where substring is found between begin index and end index.
<b>format(value)</b>	It returns a formatted version of S, using the passed value.
<b>index(subsring, beginIndex, endIndex)</b>	It throws an exception if string is not found. It works same as find() method.

<b>isalnum()</b>	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false.
<b>isalpha()</b>	It returns true if all the characters are alphabets and there is at least one character, otherwise False.
<b>isdecimal()</b>	It returns true if all the characters of the string are decimals.
<b>isdigit()</b>	It returns true if all the characters are digits and there is at least one character, otherwise False.
<b>isidentifier()</b>	It returns true if the string is the valid identifier.
<b>islower()</b>	It returns true if the characters of a string are in lower case, otherwise false.
<b>isnumeric()</b>	It returns true if the string contains only numeric characters.
<b>isprintable()</b>	It returns true if all the characters of s are printable or s is empty, false otherwise.
<b>isupper()</b>	It returns false if characters of a string are in Upper case, otherwise False.
<b>isspace()</b>	It returns true if the characters of a string are white-space, otherwise false.
<b>istitle()</b>	It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case.
<b>isupper()</b>	It returns true if all the characters of the string(if exists) is true otherwise it returns false.
<b>join(seq)</b>	It merges the strings representation of the given sequence.
<b>len(string)</b>	It returns the length of a string.
<b>ljust(width[,fillchar])</b>	It returns the space padded strings with the original string left justified to the given width.
<b>lower()</b>	It converts all the characters of a string to Lower case.
<b>lstrip()</b>	It removes all leading whitespaces of a string and can also be used to remove particular character from leading.
<b>partition()</b>	It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
<b>maketrans()</b>	It returns a translation table to be used in translate function.
<b>replace(old,new[,count])</b>	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
<b>rfind(str,beg=0,end=len(str))</b>	It is similar to find but it traverses the string in backward direction.
<b>rindex(str,beg=0,end=len(str))</b>	It is same as index but it traverses the string in backward direction.
<b>rjust(width[,fillchar])</b>	Returns a space padded string having original string right justified to the number of characters specified.
<b>rstrip()</b>	It removes all trailing whitespace of a string and can also be used to remove particular character from trailing.
<b>rsplit(sep=None, maxsplit = -1)</b>	It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space.
<b>split(str,num=string.count(str))</b>	Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter.

<b>splitlines(num=string.count('\n'))</b>	It returns the list of strings at each line with newline removed.
<b>startswith(str,beg=0,end=len(str))</b>	It returns a Boolean value if the string starts with given str between begin and end.
<b>strip([chars])</b>	It is used to perform lstrip() and rstrip() on the string.
<b>swapcase()</b>	It inverts case of all characters in a string.
<b>title()</b>	It is used to convert the string into the title-case i.e., The string meEruT will be converted to Meerut.
<b>translate(table,deletechars = "")</b>	It translates the string according to the translation table passed in the function .
<b>upper()</b>	It converts all the characters of a string to Upper Case.
<b>zfill(width)</b>	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).

## The Format Method

str.format() is one of the *string formatting methods* in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

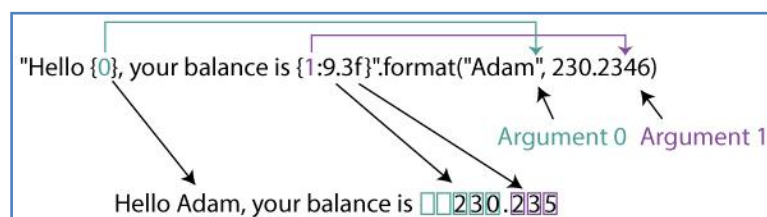
Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces { } into a string and calling the str.format(). The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.

**Syntax** : { }.format(value)

### Parameters

(value) : Can be an integer, floating point numeric constant, string, characters or even variables.

**Returntype** : Returns a formatted string with the value passed as parameter in the placeholder position.



Parameter	Description
<b>value</b>	A value of any format
<b>format</b>	The format you want to format the value into. Legal values: '<' - Left aligns the result (within the available space) '>' - Right aligns the result (within the available space) '^' - Center aligns the result (within the available space) '=' - Places the sign to the left most position '+' - Use a plus sign to indicate if the result is positive or negative



```
'-' - Use a minus sign for negative values only
' ' - Use a leading space for positive numbers
',' - Use a comma as a thousand separator
'_' - Use an underscore as a thousand separator
'b' - Binary format
'c' - Converts the value into the corresponding unicode character
'd' - Decimal format
'e' - Scientific format, with a lower case e
'E' - Scientific format, with an upper case E
'f' - Fix point number format
'F' - Fix point number format, upper case
'g' - General format
'G' - General format (using a upper case E for scientific notations)
'o' - Octal format
'x' - Hex format, lower case
'X' - Hex format, upper case
'n' - Number format
'%' - Percentage format
```

## String Slices

The slice() function returns a slice object. A slice object is used to specify how to slice a sequence. You can specify where to start the slicing, and where to end. You can also specify the step, which allows you to e.g. slice only every other item.

Syntax  
 slice(start, end, step)

Parameter Values

Parameter	Description
<b>start</b>	Optional. An integer number specifying at which position to start the slicing. Default is 0
<b>end</b>	An integer number specifying at which position to end the slicing
<b>step</b>	Optional. An integer number specifying the step of the slicing. Default is 1

Create a tuple and a slice object. Use the step parameter to return every third item:

```
a = ("a", "b", "c", "d", "e", "f", "g", "h")
x = slice(0, 8, 3)
print(a[x])
```

Output  
 ('a', 'd', 'g')



## String Operators

Operator	Description	Operation
+	Concatenates (joins) string1 and string2	string1 + string2
*	Repeats the string for as many times as specified by x	string * x
[]	Slice — Returns the character from the index provided at x.	string[x]
[:]	Range Slice — Returns the characters from the range provided at x:y.	string[x:y]
in	Membership — Returns True if x exists in the string. Can be multiple characters.	x in string
not in	Membership — Returns True if x does not exist in the string. Can be multiple characters.	x not in string
r	Suppresses an escape sequence (\x) so that it is actually rendered. In other words, it prevents the escape character from being an escape character.	r"\x"
%	Performs string formatting. It can be used as a placeholder for another value to be inserted into the string. The % symbol is a prefix to another character (x) which defines the type of value to be inserted. The value to be inserted is listed at the end of the string after another % character.	%x

Character	Description
%c	Character.
%s	String conversion via str() prior to formatting.
%i	Signed decimal integer.
%d	Signed decimal integer.
%u	Unsigned decimal integer.
%o	Octal integer.
%x	Hexadecimal integer using lowercase letters.
%X	Hexadecimal integer using uppercase letters.
%e	Exponential notation with lowercase e.
%E	Exponential notation with uppercase e.
%f	Floating point real number.
%g	The shorter of %f and %e.
%G	The shorter of %f and %E.

## Numeric Data Types

In Python, number data types are used to store numeric values. There are four different numerical types in Python:

1. **int** (plain integers): this one is pretty standard -- plain integers are just positive or negative whole numbers.
2. **long** (long integers): long integers are integers of infinite size. They look just like plain integers except they're followed by the letter "L" (ex: 150L).
3. **float** (floating point real values): floats represent real numbers, but are written with decimal points (or scientific notation) to divide the whole number into fractional parts.
4. **complex** (complex numbers): represented by the formula  $a + bj$ , where  $a$  and  $b$  are floats, and  $J$  is the square root of  $-1$  (the result of which is an imaginary number). Complex numbers are used sparingly in Python.

Applying the function call operator () to a numeric type creates an instance of that type. For example: calling int(x) will convert x to a plain integer. This can also be used with the long, float, and complex types.

## Conversions

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. **Implicit Type Conversion**
2. **Explicit Type Conversion**

### Implicit Type Conversion:

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

### Example

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo
print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))
print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

### Output

```
datatype of num_int: <class 'int'>
datatype of num_flo: <class 'float'>
Value of num_new: 124.23
datatype of num_new: <class 'float'>
```

### Example

```
num_int = 123
num_str = "456"
print("Data type of num_int:",type(num_int))
print("Data type of num_str:",type(num_str))
print(num_int+num_str)
```

### Output

```
Data type of num_int: <class 'int'>
Data type of num_str: <class 'str'>
Traceback (most recent call last):
  File "python", line 7, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the above program,

- We add two variable num\_int and num\_str.
- As we can see from the output, we got typeerror. Python is not able use Implicit Conversion in such condition.
- However Python has the solution for this type of situation which is know as Explicit Conversion.

**Explicit Type Conversion:**

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like `int()`, `float()`, `str()`, etc to perform explicit type conversion.

This type conversion is also called typecasting because the user casts (change) the data type of the objects. Typecasting can be done by assigning the required data type function to the expression.

**Example**

```
num_int = 123
num_str = "456"
print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))
num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))
num_sum = num_int + num_str
print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

**Output**

```
Data type of num_int: <class 'int'>
Data type of num_str before Type Casting: <class 'str'>
Data type of num_str after Type Casting: <class 'int'>
Sum of num_int and num_str: 579
Data type of the sum: <class 'int'>
```

In above program,

- We add `num_str` and `num_int` variable.
- We converted `num_str` from string(higher) to integer(lower) type using `int()` function to perform the addition.
- After converting `num_str` to a integer value Python is able to add these two variable.
- We got the `num_sum` value and data type to be integer.

Key Points to Remember:

1. Type Conversion is the conversion of object from one data type to another data type.
2. Implicit Type Conversion is automatically performed by the Python interpreter.
3. Python avoids the loss of data in Implicit Type Conversion.
4. Explicit Type Conversion is also called Type Casting, the data types of object are converted using predefined function by user.
5. In Type Casting loss of data may occur as we enforce the object to specific data type.

## Simple Input and Output

Python provides numerous built-in functions that are readily available to us at the Python prompt.

Some of the functions like `input()` and `print()` are widely used for standard input and output operations respectively.

### Python Output Using `print()` function.

We use the `print()` function to output data to the standard output device (screen). We can also output data to a file, but this will be discussed later. An example use is given below.

```
print('This sentence is output to the screen')
Output: This sentence is output to the screen
a = 5
print('The value of a is', a)
Output: The value of a is 5
```

In the second `print()` statement, we can notice that a space was added between the string and the value of variable `a`. This is by default, but we can change it. The actual syntax of the `print()` function is

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Here, `objects` is the value(s) to be printed. The `sep` separator is used between the values. It defaults into a space character. After all values are printed, `end` is printed. It defaults into a new line. The `file` is the object where the values are printed and its default value is `sys.stdout` (screen).

### Example

```
print(1,2,3,4)
# Output: 1 2 3 4
print(1,2,3,4,sep='*')
# Output: 1*2*3*4
print(1,2,3,4,sep='#',end='&')
# Output: 1#2#3#4&
```

## Python Input

Up till now, our programs were static. The value of variables were defined or hard coded into the source code. To allow flexibility we might want to take the input from the user. In Python, we have the `input()` function to allow this. The syntax for `input()` is

### `input([prompt])`

where `prompt` is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use `int()` or `float()` functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

This same operation can be performed using the `eval()` function. But it takes it further. It can evaluate even expressions, provided the input is a string

```
>>> int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2+3'
>>> eval('2+3')
5
```

## Control Flow and Syntax

A program's control flow is the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

### if Statement

Often, you need to execute some statements only if some condition holds, or choose statements to execute depending on several mutually exclusive conditions. The Python compound statement `if`, which uses `if`, `elif`, and `else` clauses, lets you conditionally execute blocks of statements. Here's the syntax for the `if` statement:

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

The `elif` and `else` clauses are optional. Note that unlike some languages, Python does not have a `switch` statement, so you must use `if`, `elif`, and `else` for all conditional processing.

**Here's a typical `if` statement:**

```
if x < 0: print "x is negative"
elif x % 2: print "x is positive and odd"
else: print "x is even and non-negative"
```

### The while Statement

The while statement in Python supports repeated execution of a statement or block of statements that is controlled by a conditional expression. Here's the syntax for the while statement:

```
while expression:  
    statement(s)
```

A while statement can also include an else clause and break and continue statements, as we'll discuss shortly. Here's a typical while statement:

```
count = 0  
while x > 0:  
    x = x // 2    # truncating division  
    count += 1  
print "The approximate log2 is", count
```

First, *expression*, which is known as the loop condition, is evaluated. If condition is false, while statement ends. If the loop condition is satisfied, the statement or statements that comprise the loop body are executed. When the loop body finishes executing, the loop condition is evaluated again, to see if another iteration should be performed. This process continues until the loop condition is false, at which point the while statement ends.

The loop body should contain code that eventually makes the loop condition false, or the loop will never end unless an exception is raised or the loop body executes a break statement. A loop that is in a function's body also ends if a return statement executes in the loop body, as the whole function ends in this case.

### The for Statement

The for statement in Python supports repeated execution of a statement or block of statements that is controlled by an iterable expression. Here's the syntax for the for statement:

```
for target in iterable:  
    statement(s)
```

Note that the in keyword is part of the syntax of the for statement and is functionally unrelated to the in operator used for membership testing. A for statement can also include an else clause and break and continue statements, as we'll discuss shortly.

Here's a typical for statement:

```
for letter in "ciao":  
    print "give me a", letter, "..."
```

*iterable* may be any Python expression suitable as an argument to built-in function iter, which returns an iterator object (explained in detail in the next section). *target* is normally an identifier that names the control variable of the loop; the for statement successively rebinds this variable to each item of the iterator, in order. The statement or state-



ments that comprise the loopbody execute once for each item in *iterable* (unless the loop ends because an exception is raised or a break or return statement is executed).

### range

Looping over a sequence of integers is a common task, so Python provides built-in functions range and xrange to generate and return integer sequences.

```
for i in xrange(n):
    statement(s)
```

Range( *x* ) returns a list whose items are consecutive integers from 0(included) up to *x* (excluded). range( *x,y* ) returns a list whose items are consecutive integers from *x* (included) up to *y* (excluded). The result is the empty list if *x* is greater than or equal to *y*. range( *x,y,step* ) returns a list of integers from *x* (included) up to *y* (excluded), such that the difference between each two adjacent items in the list is *step*. If *step* is less than 0, range counts down from *x* to *y*. range returns the empty list when *x* is greater than or equal to *y* and *step* is greater than 0, or when *x* is less than or equal to *y* and *step* is less than 0. If *step* equals 0, range raises an exception.

### The break Statement

The break statement is allowed only inside a loop body. When break executes, the loop terminates. If a loop is nested inside other loops, break terminates only the innermost nested loop. In practical use, a break statement is usually inside some clause of an if statement in the loop body so that it executes conditionally.

### The continue Statement

The continue statement is allowed only inside a loop body. When continue executes, the current iteration of the loop body terminates, and execution continues with the next iteration of the loop. In practical use, a continuestatement is usually inside some clause of an if statement in the loop body so that it executes conditionally.

### The pass Statement

The body of a Python compound statement cannot be empty. It must contain at least one statement. The pass statement, which performs no action, can be used as a placeholder when a statement is syntactically required but you have nothing specific to do.

## Relational Operators

Relational operators are used to establish some sort of relationship between the two operands. Some of the relevant examples could be less than, greater than or equal to operators. Python language is capable of understanding these types of operators and accordingly return the output, which can be either True or False.

Oper.	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.



>	If the value of left operand is greater than the value of right operand, (a > b) is not true. then condition becomes true.
<	If the value of left operand is less than the value of right operand, then (a < b) is true. condition becomes true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. (a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true. (a <= b) is true.

## Logical Operators

Logical operators, as the name suggests are used in logical expressions where the operands are either True or False. The operands in a logical expression can be expressions which return True or False upon evaluation. There are three basic types of logical operators:

Operator	Description	Example
<b>and Logical AND</b>	If both the operands are true then condition becomes true.	(a and b) is true.
<b>or Logical OR</b>	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
<b>not Logical NOT</b>	Used to reverse the logical state of its operand.	Not(a and b) is false.

## Bit Wise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows –

a = 0011 1100
b = 0000 1101
-----
a&b = 0000 1100
a b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

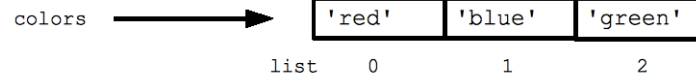
There are following Bitwise operators supported by Python language.

Operator	Description	Example
<b>&amp; Binary AND</b>	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
<b>  Binary OR</b>	It copies a bit if it exists in either operand.	(a   b) = 61 (means 0011 1101)
<b>^ Binary XOR</b>	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
<b>~ Binary Ones Complement</b>	It is unary and has the effect of 'flipping' bits.	(~a ) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<b>&lt;&lt; Binary Left Shift</b>	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
<b>&gt;&gt; Binary Right Shift</b>	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

## Lists

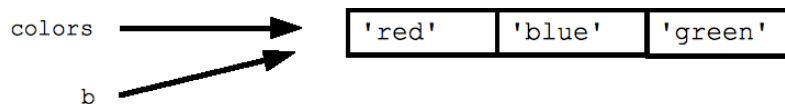
Python has a great built-in list type named "list". List literals are written within square brackets [ ]. Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0.

```
colors = ['red', 'blue', 'green']
print colors[0] ## red
print colors[2] ## green
print len(colors) ## 3
```



Assignment with an = on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.

```
b = colors ## Does not copy the list
```



The "empty list" is just an empty pair of brackets [ ]. The '+' works to append two lists, so [1, 2] + [3, 4] yields [1, 2, 3, 4] (this is just like + with strings).

### List Methods

Here are some other common list methods.

- list.append(elem) -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- list.remove(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)
- list.sort() -- sorts the list in place (does not return it). (The sorted() function shown later is preferred.)
- list.reverse() -- reverses the list in place (does not return it)
- list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Notice that these are *\*methods\** on a list object, while `len()` is a function that takes the list (or string or whatever) as an argument.

```
list = ['larry', 'curly', 'moe']
list.append('shemp')    ## append elem at end
list.insert(0, 'xxx')  ## insert elem at index 0
list.extend(['yyy', 'zzz']) ## add list of elems at end
print list    ## ['xxx', 'larry', 'curly', 'moe', 'shemp', 'yyy', 'zzz']
print list.index('curly')    ## 2
list.remove('curly')    ## search and remove that element
list.pop(1)            ## removes and returns 'larry'
print list    ## ['xxx', 'moe', 'shemp', 'yyy', 'zzz']
```

**Common error:** note that the above methods do not *\*return\** the modified list, they just modify the original list.

```
list = [1, 2, 3]
print list.append(4)    ## NO, does not work, append() returns None
## Correct pattern:
list.append(4)
print list    ## [1, 2, 3, 4]
```

### List Build Up

One common pattern is to start a list as the empty list `[]`, then use `append()` or `extend()` to add elements to it:

```
list = []    ## Start as the empty list
list.append('a')    ## Use append() to add elements
list.append('b')
```

### List Slices

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

```
list = ['a', 'b', 'c', 'd']
print list[1:-1]    ## ['b', 'c']
list[0:2] = 'z'    ## replace ['a', 'b'] with ['z']
print list    ## ['z', 'c', 'd']
```

## Tuples

A tuple is a fixed size grouping of elements, such as an (x, y) co-ordinate. Tuples are like lists, except they are immutable and do not change size (tuples are not strictly immutable since one of the contained elements could be mutable). Tuples play a sort of "struct" role in Python -- a convenient way to pass around a little logical, fixed size bundle of values. A function that needs to return multiple values can just return a tuple of the values. For example, if I wanted to have a list of 3-d coordinates, the natural python representation would be a list of tuples, where each tuple is size 3 holding one (x, y, z) group.

To create a tuple, just list the values within parenthesis separated by commas. The "empty" tuple is just an empty pair of parenthesis. Accessing the elements in a tuple is just like a list -- len(), [ ], for, in, etc. all work the same.

```
tuple = (1, 2, 'hi')
print len(tuple) ## 3
print tuple[2] ## hi
tuple[2] = 'bye' ## NO, tuples cannot be changed
tuple = (1, 2, 'bye') ## this works
```

To create a size-1 tuple, the lone element must be followed by a comma.

```
tuple = ('hi',) ## size-1 tuple
```

It's a funny case in the syntax, but the comma is necessary to distinguish the tuple from the ordinary case of putting an expression in parentheses. In some cases you can omit the parenthesis and Python will see from the commas that you intend a tuple.

Assigning a tuple to an identically sized tuple of variable names assigns all the corresponding values. If the tuples are not the same size, it throws an error. This feature works for lists too.

```
(x, y, z) = (42, 13, "hike")
print z ## hike
(err_string, err_code) = Foo() ## Foo() returns a length-2 tuple
```

## Sets

The set in python can be defined as the unordered collection of various items enclosed within the curly braces. The elements of the set can not be duplicate. The elements of the python set must be immutable.

Unlike other collections in python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together or we can get the list of elements by looping through the set.

### Creating a set

The set can be created by enclosing the comma separated items with the curly braces. Python also provides the set method which can be used to create the set by the passed sequence.

**Example 1: using curly braces**

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
print(Days)
print(type(Days))
print("looping through the set elements ... ")
for i in Days:
    print(i)
```

**Output:**

```
{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}
looping through the set elements ...
Friday
Tuesday
Monday
Saturday
Thursday
Sunday
Wednesday
```

**Example : using set() method**

```
Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
print(Days)
print(type(Days))
print("looping through the set elements ... ")
for i in Days:
    print(i)
```

**Output:**

```
{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}
looping through the set elements ...
Friday
Wednesday
Thursday
Saturday
Monday
Tuesday
Sunday
```

**Python Set operations**

In the previous example, we have discussed about how the set is created in python. However, we can perform various mathematical operations on python sets like union, intersection, difference, etc.

**Adding items to the set**

Python provides the add() method which can be used to add some particular item to the set.

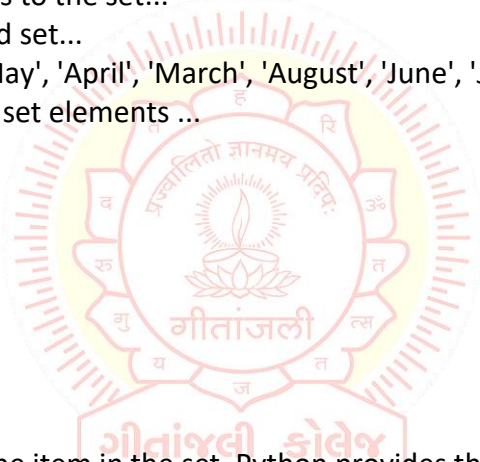
Consider the following example.

**Example:**

```
Months = set(["January", "February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nAdding other months to the set...");
Months.add("July");
Months.add("August");
print("\nPrinting the modified set...");
print(Months)
print("\nlooping through the set elements ... ")
for i in Months:
    print(i)
```

**Output:**

```
printing the original set ...
{'February', 'May', 'April', 'March', 'June', 'January'}
Adding other months to the set...
Printing the modified set...
{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}
looping through the set elements ...
February
July
May
April
March
August
June
January
To add more than one item in the set, Python provides the update() method.
```



**Example**

```
Months = set(["January", "February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nupdating the original set ... ")
Months.update(["July", "August", "September", "October"]);
print("\nprinting the modified set ... ")
print(Months);
```

**Output:**

```
printing the original set ...
{'January', 'February', 'April', 'May', 'June', 'March'}
updating the original set ...
printing the modified set ...
{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}
```

**Removing items from the set**

Python provides discard() method which can be used to remove the items from the set.

**Example**

```
Months = set(["January", "February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving some months from the set...");
Months.discard("January");
Months.discard("May");
print("\nPrinting the modified set...");
print(Months)
print("\nlooping through the set elements ... ")
for i in Months:
    print(i)
```

**Output:**

```
printing the original set ...
{'February', 'January', 'March', 'April', 'June', 'May'}
Removing some months from the set...
Printing the modified set...
{'February', 'March', 'April', 'June'}
looping through the set elements ...
February
March
April
June
```

Python also provide the remove() method to remove the items from the set. Consider the following example to remove the items using remove() method.

**Example**

```
Months = set(["January", "February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving some months from the set...");
Months.remove("January");
Months.remove("May");
print("\nPrinting the modified set...");
print(Months)
```

**Output:**

```
printing the original set ...
{'February', 'June', 'April', 'May', 'January', 'March'}
Removing some months from the set...
Printing the modified set...
{'February', 'June', 'April', 'March'}
```



We can also use the **pop()** method to remove the item. However, this method will always remove the last item.

Consider the following example to remove the last item from the set.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving some months from the set...");
Months.pop();
Months.pop();
print("\nPrinting the modified set...");
print(Months)
```

#### Output:

```
printing the original set ...
{'June', 'January', 'May', 'April', 'February', 'March'}
Removing some months from the set...
Printing the modified set...
{'May', 'April', 'February', 'March'}
```

Python provides the **clear()** method to remove all the items from the set.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving all the items from the set...");
Months.clear()
print("\nPrinting the modified set...")
print(Months)
```

#### Output:

```
printing the original set ...
{'January', 'May', 'June', 'April', 'March', 'February'}
Removing all the items from the set...
Printing the modified set...
set()
```

#### Difference between discard() and remove()

Despite the fact that **discard()** and **remove()** method both perform the same task, There is one main difference between **discard()** and **remove()**.

If the key to be deleted from the set using **discard()** doesn't exist in the set, the python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using **remove()** doesn't exist in the set, the python will give the error.

Consider the following example.

**Example**

```
Months = set(["January", "February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving items through discard() method...");
Months.discard("Feb"); #will not give an error although the key feb is not available in
the set
print("\nprinting the modified set...")
print(Months)
print("\nRemoving items through remove() method...");
Months.remove("Jan") #will give an error as the key jan is not available in the set.
print("\nPrinting the modified set...")
print(Months)
```

**Output:**

```
printing the original set ...
{'March', 'January', 'April', 'June', 'February', 'May'}
Removing items through discard() method...
printing the modified set...
{'March', 'January', 'April', 'June', 'February', 'May'}
```

**Removing items through remove() method...**

```
Traceback (most recent call last):
  File "set.py", line 9, in
    Months.remove("Jan")
KeyError: 'Jan'
```

**Union of two Sets**

The union of two sets are calculated by using the or (**|**) operator. The union of the two sets contains the all the items that are present in both the sets.

Consider the following example to calculate the union of two sets.

**Example :** using union | operator

```
Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
Days2 = {"Friday", "Saturday", "Sunday"}
print(Days1 | Days2) #printing the union of the sets
```

**Output:**

```
{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 'Thursday'}
```

Python also provides the union() method which can also be used to calculate the union of two sets. Consider the following example.

**Example** : using union() method

```
Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
Days2 = {"Friday","Saturday","Sunday"}
print(Days1.union(Days2)) #printing the union of the sets
```

**Output:**

```
{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 'Saturday'}
```

**Intersection of two sets**

The & (intersection) operator is used to calculate the intersection of the two sets in python. The intersection of the two sets are given as the set of the elements that common in both sets.

Consider the following example.

**Example** : using & operator

```
set1 = {"Ayush","John", "David", "Martin"}
set2 = {"Steve","Milan","David", "Martin"}
print(set1&set2) #prints the intersection of the two sets
```

**Output:** {'Martin', 'David'}**Example** : using intersection() method

```
set1 = {"Ayush","John", "David", "Martin"}
set2 = {"Steve","Milan","David", "Martin"}
print(set1.intersection(set2)) #prints the intersection of the two sets
```

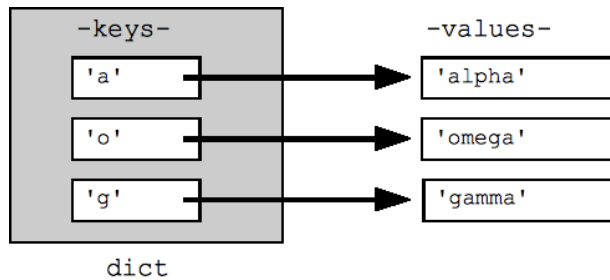
**Output:** {'Martin', 'David'}**Dictionaries**

Python's efficient key/value hash table structure is called a "dict". The contents of a dict can be written as a series of key:value pairs within braces { }, e.g. dict = {key1:value1, key2:value2, ... }. The "empty dict" is just an empty pair of curly braces {}.

Looking up or setting a value in a dict uses square brackets, e.g. dict['foo'] looks up the value under the key 'foo'. Strings, numbers, and tuples work as keys, and any type can be a value. Other types may or may not work correctly as keys (strings and tuples work cleanly since they are immutable). Looking up a value which is not in the dict throws a KeyError -- use "in" to check if the key is in the dict, or use dict.get(key) which returns the value or None if the key is not present (or get(key, not-found) allows you to specify what value to return in the not-found case).

```
## Can build up a dict by starting with the the empty dict {}
## and storing key/value pairs into the dict like this:
## dict[key] = value-for-that-key
dict = {}
dict['a'] = 'alpha'
dict['g'] = 'gamma'
```

```
dict['o'] = 'omega'
print dict ## {'a': 'alpha', 'o': 'omega', 'g': 'gamma'}
print dict['a'] ## Simple lookup, returns 'alpha'
dict['a'] = 6 ## Put new key/value into dict
'a' in dict ## True
## print dict['z'] ## Throws KeyError
if 'z' in dict: print dict['z'] ## Avoid KeyError
print dict.get('z') ## None (instead of KeyError)
```



## Dict Formatting

The % operator works conveniently to substitute values from a dict into a string by name:

```
hash = {}
hash['word'] = 'garfield'
hash['count'] = 42
s = 'I want %(count)d copies of %(word)s' % hash # %d for int, %s for string
# 'I want 42 copies of garfield'
```

## Del

The "del" operator does deletions. In the simplest case, it can remove the definition of a variable, as if that variable had not been defined. Del can also be used on list elements or slices to delete that part of the list and to delete entries from a dictionary.

```
var = 6
del var # var no more!
list = ['a', 'b', 'c', 'd']
del list[0] ## Delete first element
del list[-2:] ## Delete last two elements
print list ## ['b']
dict = {'a':1, 'b':2, 'c':3}
del dict['b'] ## Delete 'b' entry
print dict ## {'a':1, 'c':3}
```

## Sorting Dictionaries

The dict (dictionary) class object in Python is a very versatile and useful container type, able to store a collection of values and retrieve them via keys. The values can be objects of any type (dictionaries can even be nested with other dictionaries) and the keys can be any object so long as it's *hashable*, meaning basically that it is immutable (so strings are not the only valid keys, but mutable objects like lists can never be used as keys). Unlike Py-

thon lists or tuples, the key and value pairs in dict objects are not in any particular order, which means we can have a dict like this:

```
numbers = {'first': 1, 'second': 2, 'third': 3, 'Fourth': 4}
```

Although the key-value pairs are in a certain order in the instantiation statement, by calling the *list* method on it (which will create a list from its keys) we can easily see they aren't stored in that order:

```
>>> list(numbers)
['second', 'Fourth', 'third', 'first']
```

### Sorting Python dictionaries by Keys

If we want to order or sort the dictionary objects by their keys, the simplest way to do so is by Python's built-in *sorted* method, which will take any iterable and return a list of the values which has been sorted (in ascending order by default). There is no class method for sorting dictionaries as there is for lists, however the sorted method works the same exact way. Here's what it does with our dictionary:

```
# This is the same as calling sorted(numbers.keys())
>>> sorted(numbers)
['Fourth', 'first', 'second', 'third']
```

We can see this method has given us a list of the keys in ascending order, and in almost alphabetical order, depending on what we define as "alphabetical." Also notice that we sorted its list of keys *by its keys* — if we want to sort its list of *values* by its keys, or its list of *keys* by its values, we'd have to change the way we use the sorted method. We'll look at these different aspects of sorted in a bit.

### Sorting Python dictionaries by Values

In the same way as we did with the keys, we can use *sorted* to sort the Python dictionary by its values:

```
# We have to call numbers.values() here
>>> sorted(numbers.values())
[1, 2, 3, 4]
```

This is the list of values in the default order, in ascending order. These are very simple examples so let's now examine some slightly more complex situations where we are sorting our dict object.

### Custom sorting algorithms with Python dictionaries

If we simply give the sorted method the dictionary's keys/values as an argument it will perform a simple sort, but by utilizing its other arguments (i.e. key and reverse) we can get it to perform more complex sorts.

The key argument (not to be confused with the dictionary's keys) for sorted allows us to define specific functions to use when sorting the items, as an *iterator* (in our dict object). In both examples above the keys and values were both the items to sort and the items used for comparison, but if we want to sort our dict *keys* using our dict *values*, then we would tell sorted to do that via its key argument. Such as follows:

```
# Use the __getitem__ method as the key function
>>> sorted(numbers, key=numbers.__getitem__)
# In order of sorted values: [1, 2, 3, 4]
['first', 'second', 'third', 'Fourth']
```

With this statement we told sorted to sort the numbers dict (its keys), and to sort them by using numbers' class method for retrieving values — essentially we told it "for every key in *numbers*, use the corresponding value in *numbers* for comparison to sort it."

We can also sort the *values* in numbers by its keys, but using the key argument would be more complicated (there is no dictionary method to return a key by using a certain value, as with the list.index method). Instead we can use a list comprehension to keep it simple:

```
# Uses the first element of each tuple to compare
>>> [value for (key, value) in sorted(numbers.items())]
[4, 1, 2, 3]

# In order of sorted keys: ['Fourth', 'first', 'second', 'third']
```

Now the other argument to consider is the reverse argument. If this is True, the order will be reversed (descending), otherwise if it's False it will be in the default (ascending) order, it's as simple as that. For example, as with the two previous sorts:

```
>>> sorted(numbers, key=numbers.__getitem__, reverse=True)
['Fourth', 'third', 'second', 'first']
>>> [value for (key, value) in sorted(numbers.items(), reverse=True)]
[3, 2, 1, 4]
```

These sorts are still fairly simple, but let's look at some special algorithms we might use with strings or numbers to sort our dictionary.

## Copying Collections

In Python, we use = operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object. Let's take an example where we create a list named old\_list and pass an object reference to new\_list using = operator.

**Example :** Copy using = operator

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list
new_list[2][2] = 9
print('Old List:', old_list)
print('ID of Old List:', id(old_list))
print('New List:', new_list)
print('ID of New List:', id(new_list))
```



When we run above program, the output will be:

```
Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of Old List: 140673303268168
New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of New List: 140673303268168
```

As you can see from the output both variables `old_list` and `new_list` shares the same id i.e 140673303268168.

So, if you want to modify any values in `new_list` or `old_list`, the change is visible in both. Essentially, sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

1. **Shallow Copy**
2. **Deep Copy**

### Shallow Copy

A shallow copy creates a new object which stores the reference of the original elements. So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.

**Example :** Create a copy using shallow copy

```
import copy
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_list = copy.copy(old_list)
print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program , the output will be:

```
Old list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
New list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In above program, we created a nested list and then shallow copy it using `copy()` method. This means it will create new and independent object with same content. To verify this, we print the both `old_list` and `new_list`.

To confirm that `new_list` is different from `old_list`, we try to add new nested object to original and check it.

**Example :** Adding [4, 4, 4] to `old_list`, using shallow copy

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list.append([4, 4, 4])
print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```



In the above program, we created a shallow copy of `old_list`. The `new_list` contains references to original nested objects stored in `old_list`. Then we add the new list i.e [4, 4, 4] into `old_list`. This new sublist was not copied in `new_list`.

### Deep Copy

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements. However, we are going to create deep copy using `deepcopy()` function present in `copy` module. The deep copy creates independent copy of original object and all its nested objects.

#### Example 5: Copying a list using `deepcopy()`

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, we use `deepcopy()` function to create copy which looks similar. However, if you make changes to any nested objects in original object `old_list`, you'll see no changes to the copy `new_list`.

#### Example 6: Adding a new nested object in the list using Deep copy

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
old_list[1][0] = 'BB'
print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, when we assign a new value to `old_list`, we can see only the `old_list` is modified. This means, both the `old_list` and the `new_list` are independent. This is because the `old_list` was recursively copied, which is true for all its nested objects.

## Defining Your Own Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions. These functions are called *user-defined functions*.

### Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword `def` followed by the function name and parentheses `( )`.
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon `(:)` and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

### Parameters

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

#### Example:

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):  
    "This prints a passed string into this function"  
    print str  
    return
```

### Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call `printme()` function –

```
# Function definition is here  
def printme( str ):  
    "This prints a passed string into this function"  
    print str
```

```

return
# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")

```

When the above code is executed, it produces the following result –

```

I'm first call to user defined function!
Again second call to the same function

```

## Function Documentation

Function doc can be described as soon as describing function. It can be inserted between triple inverted commas.

```

# Function definition is here
def printme( str ):
    """This Function is used to pass the string inside the function."""
    "This prints a passed string into this function"
    print str
    return

```

Above function is documented into python's help.

## Keyword and Optional Parameters

### Keyword Arguments

While making a function call, you can mention the parameter name and assign a value to it, with param\_name=value syntax, to explicitly instruct the function about the variable assignments. In this case, the arguments can be passed in any order.

```

>>> def myFunction(a, b, c):
...     print "Value of 'a' = " + str(a)
...     print "Value of 'b' = " + str(b)
...     print "Value of 'c' = " + str(c)
...
>>> myFunction(b=20, c=50, a=10)
Value of 'a' = 10
Value of 'b' = 20
Value of 'c' = 50
>>> myFunction(c=20, a=50, b=10)
Value of 'a' = 50
Value of 'b' = 10
Value of 'c' = 20

```

In above examples, we have assigned values to the function parameters in the function call itself. What would have happened, had we passed only two arguments, instead of three?

```
>>> myFunction(b=20, c=50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: myFunction() takes exactly 3 arguments (2 given)
```

It throws in exception, saying that 'you have provided only 2 arguments when I expect 3 from you'. But what if we want to provide only two arguments or even one, expecting that the function should display None if a value is absent? This leads us to discuss on *Defaults* or *Optional arguments*.

### Optional Arguments

With *defaults*, we can assign a default value to a function parameter. While making a function call, if the argument is not provided, the parameter takes the default value assigned to it. This not only avoids an exception, but makes the argument optional. When the values are specified, default values will be overwritten and parameters will be assigned with the values provided in the argument.

#### # Function with two optional arguments

```
>>> def welcome(name='User', country='India'):
...     print 'Hello ' + name + '! Welcome to ' + country + '!'
... 
```

#### # We specify no argument, default arguments will be taken

```
>>> welcome()
Hello User! Welcome to India.
```

#### # We do not mention 'country' here, default value will be taken

```
>>> welcome(name='Mandar')
Hello Mandar! Welcome to India.
```

#### # We overwrite both the default values, using keyword arguments

```
>>> welcome(name='Mandar', country='America')
Hello Mandar! Welcome to America.
```

#### # With non-keyword arguments - order is crucial

```
>>> welcome('Mandar', 'Russia')
Hello Mandar! Welcome to Russia.
```

#### # We specify the name in a variable

```
>>> name = 'Mandar'
>>> welcome(name, 'Australia')
Hello Mandar! Welcome to Australia.
```

In above example, we created a function welcome() that takes two arguments name and country. We assign both these variables with default values 'User' and 'India' respectively, making both of these arguments optional. If we do not specify any of the argument, it's default value will be printed. If we specify it, the default values are overwritten with the values specified. Please observe the default value 'User' getting overwritten with the value 'Mandar' and the value 'India' with 'America'.

## Passing Collections to a Function.

Passing collection to function is similar or like passing an parameter.

Example:

```
>>> listData = ['Hello','I am']
>>> listData
['Hello', 'I am']
>>> def abc(listData):
...     print(listData)
...
>>>abc(listData)
['Hello', 'I am']
```



## HTTP Client-Server Request – Response

The client-server architecture includes two major components request and response. The Django framework uses client-server architecture to implement web applications. When a client requests for a resource, a `HttpRequest` object is created and correspond view function is called that returns `HttpResponse` object. To handle request and response, Django provides `HttpRequest` and `HttpResponse` classes. Each class has it's own attributes and methods. Let's have a look at the `HttpRequest` class.

### Django HttpRequest

This class is defined in the `django.http` module and used to handle the client request. Following are the attributes of this class.

#### Django HttpRequest Attributes

Attribute	Description
<code>HttpRequest.scheme</code>	A string representing the scheme of the request (HTTP or HTTPS usually).
<code>HttpRequest.body</code>	It returns the raw HTTP request body as a byte string.
<code>HttpRequest.path</code>	It returns the full path to the requested page does not include the scheme or domain.
<code>HttpRequest.path_info</code>	It shows path info portion of the path.
<code>HttpRequest.method</code>	It shows the HTTP method used in the request.
<code>HttpRequest.encoding</code>	It shows the current encoding used to decode form submission data.
<code>HttpRequest.content_type</code>	It shows the MIME type of the request, parsed from the <code>CONTENT_TYPE</code> header.
<code>HttpRequest.content_params</code>	It returns a dictionary of key/value parameters included in the <code>CONTENT_TYPE</code> header.
<code>HttpRequest.GET</code>	It returns a dictionary-like object containing all given HTTP GET parameters.
<code>HttpRequest.POST</code>	It is a dictionary-like object containing all given HTTP POST parameters.
<code>HttpRequest.COOKIES</code>	It returns all cookies available.
<code>HttpRequest.FILES</code>	It contains all uploaded files.
<code>HttpRequest.META</code>	It shows all available Http headers.
<code>HttpRequest.resolver_match</code>	It contains an instance of <code>ResolverMatch</code> representing the resolved URL.



### Django HttpRequest Methods

Attribute	Description
<code>HttpRequest.get_host()</code>	It returns the original host of the request.
<code>HttpRequest.get_port()</code>	It returns the originating port of the request.
<code>HttpRequest.get_full_path()</code>	It returns the path, plus an appended query string, if applicable.
<code>HttpRequest.build_absolute_uri (location)</code>	It returns the absolute URI form of location.
<code>HttpRequest.get_signed_cookie (key, default=RAISE_ERROR, salt="", max_age=None)</code>	It returns a cookie value for a signed cookie, or raises a <code>django.core.signing.BadSignature</code> exception if the signature is no longer valid.
<code>HttpRequest.is_secure()</code>	It returns True if the request is secure; that is, if it was made with HTTPS.
<code>HttpRequest.is_ajax()</code>	It returns True if the request was made via an XMLHttpRequest.

### Django HttpRequest Example

```
// views.py
```

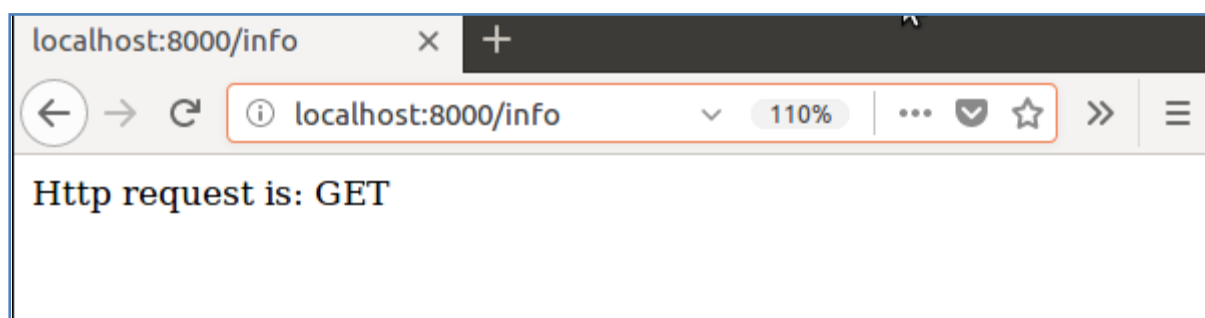
```
def methodinfo(request):
    return HttpResponse("Http request is: "+request.method)
```

```
// urls.py
```

```
path('info',views.methodinfo)
```

Start the server and get access to the browser. It shows the request method name at the browser.

Output:



## Django HttpResponse

This class is a part of **django.http** module. It is responsible for generating response corresponds to the request and back to the client.

### Django HttpResponse Attributes

Attribute	Description
<b>HttpResponse.content</b>	A bytestring representing the content, encoded from a string if necessary.
<b>HttpResponse.charset</b>	It is a string denoting the charset in which the response will be encoded.
<b>HttpResponse.status_code</b>	It is an <b>HTTP status code</b> for the response.
<b>HttpResponse.reason_phrase</b>	The HTTP reason phrase for the response.
<b>HttpResponse.streaming</b>	It is false by default.
<b>HttpResponse.closed</b>	It is True if the response has been closed.

### Django HttpResponse Methods

Method	Description
<b>HttpResponse.__init__(content="", content_type=None, status=200, reason=None, charset=None)</b>	It is used to instantiate an HttpResponse object with the given page content and content type.
<b>HttpResponse.__setitem__(header, value)</b>	It is used to set the given header name to the given value.
<b>HttpResponse.__delitem__(header)</b>	It deletes the header with the given name.
<b>HttpResponse.__getitem__(header)</b>	It returns the value for the given header name.
<b>HttpResponse.has_header(header)</b>	It returns either True or False based on a case-insensitive check for a header with the provided name.
<b>HttpResponse.setdefault(header, value)</b>	It is used to set default header.
<b>HttpResponse.write(content)</b>	It is used to create response object of file-like object.
<b>HttpResponse.flush()</b>	It is used to flush the response object.
<b>HttpResponse.tell()</b>	This method makes an HttpResponse instance a file-like object.
<b>HttpResponse.getvalue()</b>	It is used to get the value of HttpResponse.content.
<b>HttpResponse.readable()</b>	This method is used to create stream-like object of HttpResponse class.
<b>HttpResponse.seekable()</b>	It is used to make response object seekable.

We can use these methods and attributes to handle the response in the Django application.

## Concept of Web Framework and Web Application

A web framework (WF) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources, and web APIs. Web frameworks, provide a standard way to build and deploy web applications.

Web frameworks aim to automate the overhead associated with common activities performed in web development. For example, many web frameworks provide libraries for database access, templating frameworks, and session management, and they often promote code reuse. Although they often target development of dynamic web sites, they are also applicable to static websites.

### **Types of framework architectures**

Most web frameworks are based on the model–view–controller (MVC) pattern.

#### **Model–view–controller (MVC)**

Many frameworks follow the MVC architectural pattern to separate the data model with business rules from the user interface. This is generally considered a good practice as it modularizes code, promotes code reuse, and allows multiple interfaces to be applied. In web applications, this permits different views to be presented, such as web pages for humans, and web service interfaces for remote applications.

#### **Push-based vs. pull-based**

Most MVC frameworks follow a push-based architecture also called "action-based". These frameworks use actions that do the required processing, and then "push" the data to the view layer to render the results. Django, Ruby on Rails, Symfony, Spring MVC, Stripes, CodeIgniter are good examples of this architecture.

#### **Three-tier organization**

In three-tier organization, applications are structured around three physical tiers: client, application, and database. The database is normally an RDBMS. The application contains the business logic, running on a server and communicates with the client using HTTP. The client on web applications is a web browser that runs HTML generated by the application layer.

#### **Framework applications :**

Frameworks are built to support the construction of internet applications based on a single programming language, ranging in focus from general purpose tools such as Zend Framework and Ruby on Rails, which augment the capabilities of a specific language, to native-language programmable packages built around a specific user application, such as Content Management systems, some mobile development tools and some portal tools.

#### **General-purpose website frameworks:**

Web frameworks must function according to the architectural rules of browsers and web protocols such as HTTP, which is stateless. Webpages are served up by a server and can then be modified by the browser using JavaScript. Either approach has its advantages and disadvantages.

Server-side page changes typically require that the page be refreshed, but allow any language to be used and more computing power to be utilized. Client-side changes allow the page to be updated in small chunks which feels like a desktop application, but are limited to JavaScript and run in the user's browser, which may have limited computing power. Some mix of the two is typically used

## Introduction to Django

Django is a web application framework written in Python programming language. It is based on MVT (Model View Template) design pattern. The Django is very demanding due to its rapid development feature. It takes less time to build application after collecting client requirement.

This framework uses a famous tag line: The web framework for perfectionists with deadlines. By using Django, we can build web applications in very less time. Django is designed in such a manner that it handles much of configure things automatically, so we can focus on application development only.

Django was design and developed by Lawrence journal world in 2003 and publicly released under BSD license in July 2005. Currently, DSF (Django Software Foundation) maintains its development and release cycle.

Django was released on 21, July 2005. Its current stable version is 2.0.3 which was released on 6 March, 2018.

### Features of Django:

#### Rapid Development

Django was designed with the intention to make a framework which takes less time to build web application. The project implementation phase is a very time taken but Django creates it rapidly.

#### Secure

Django takes security seriously and helps developers to avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery etc. Its user authentication system provides a secure way to manage user accounts and passwords.

#### Scalable

Django is scalable in nature and has ability to quickly and flexibly switch from small to large scale application project.

#### Fully loaded

Django includes various helping task modules and libraries which can be used to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds etc.

#### Versatile

Django is versatile in nature which allows it to build applications for different-different domains. Now days, Companies are using Django to build various types of applications like: content management systems, social networks sites or scientific computing platforms etc.

**Open Source**

Django is an open source web application framework. It is publicly available without cost. It can be downloaded with source code from the public repository. Open source reduces the total cost of the application development.

**Vast and Supported Community**

Django is an one of the most popular web framework. It has widely supportive community and channels to share and connect.

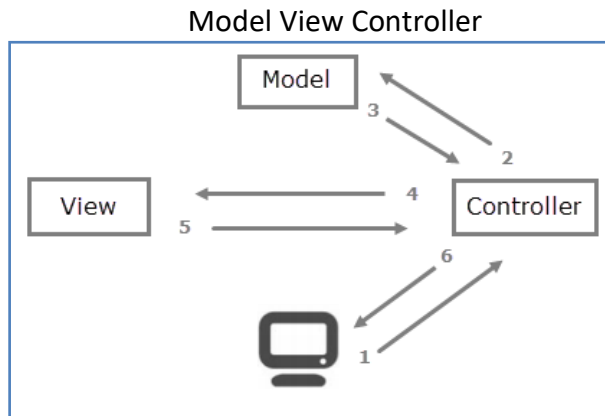
**Django is widely accepted and used by various well-known sites such as:**

- Instagram
- Mozilla
- Disqus
- Pinterest
- Bitbucket
- The Washington Times
- Nasa
- Disqus
- Knight Foundation
- MacArthur Foundation
- National Geographic
- Open Knowledge Foundation
- Open Stack



## MVC Design Pattern

The MVC pattern was created to separate business logic from representation. MVC is the most popular architecture in use today. Many popular frameworks like Ruby on Rails, Laravel, CodeIgniter and even Django uses it. The MVC architecture divides an application into the following three layers:



Here is a rundown of steps involved in an MVC blog application.

1. Web browser or client sends the request to the web server, asking the server to display a blog post.
2. The request received by the server is passed to the controller of the application.
3. The controller asks the model to fetch the blog post.
4. The model sends the blog post to the controller.
5. The controller then passes the blog post data to the view.
6. The view uses blog post data to create an HTML page.
7. At last, the controller returns the HTML content to the client.

## Django MVT

The MVT (Model View Template) is a software design pattern. It is a collection of three important components Model View and Template. The Model helps to handle database. It is a data access layer which handles the data.

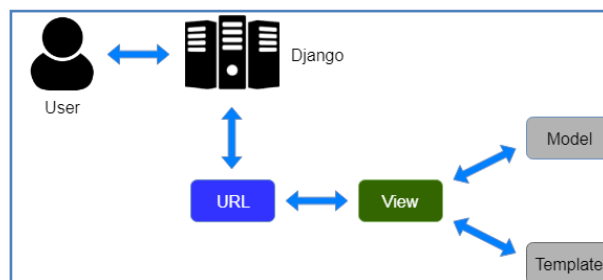
The Template is a presentation layer which handles User Interface part completely. The View is used to execute the business logic and interact with a model to carry data and renders a template.

Although Django follows MVC pattern but maintains its own conventions. So, control is handled by the framework itself.

There is no separate controller and complete application is based on Model View and Template. That's why it is called MVT application.

Here, a user **requests** for a resource to the Django, Django works as a controller and check to the available resource in URL.

If URL maps, a **view is called** that interact with model and template, it renders a template. Django responds back to the user and sends a template as a **response**.





## Django installation

### Install Python

Django is a Python web framework, thus requiring Python to be installed on your machine. At the time of writing, Python 3.5 is the latest version.

To install Python on your machine go to <https://python.org/downloads/>. The website should offer you a download button for the latest Python version. Download the executable installer and run it. Check the box next to Add Python 3.5 to PATH and then click Install Now.

After installation, open the command prompt and check that the Python version matches the version you installed by executing:

```
python --version
```

### **About pip**

pip is a package manager for Python. It makes installing and uninstalling Python packages (such as Django!) very easy. For the rest of the installation, we'll use pip to install Python packages from the command line.

To install pip on your machine, go to <https://pip.pypa.io/en/latest/installing/>, and follow the Installing with get-pip.py instructions.

### **Install virtualenv and virtualenvwrapper**

virtualenv and virtualenvwrapper provide a dedicated environment for each Django project you create. While not mandatory, this is considered a best practice and will save you time in the future when you're ready to deploy your project. Simply type:

```
pip install virtualenvwrapper-win
```

Then create a virtual environment for your project:

```
mkvirtualenv myproject
```

The virtual environment will be activated automatically and you'll see "(myproject)" next to the command prompt to designate that. If you start a new command prompt, you'll need to activate the environment again using:

```
workon myproject
```

### **Install Django**

Django can be installed easily using pip within your virtual environment. In the command prompt, ensure your virtual environment is active, and execute the following command:

```
pip install django
```

This will download and install the latest Django release.

After the installation has completed, you can verify your Django installation by executing `django-admin --version` in the command prompt.

## Setting up Database

If you plan to use Django's database API functionality, you'll need to make sure a database server is running. Django supports many different database servers and is officially supported with PostgreSQL, MySQL, Oracle and SQLite.

If you are developing a simple project or something you don't plan to deploy in a production environment, SQLite is generally the simplest option as it doesn't require running a separate server. However, SQLite has many differences from other databases, so if you are working on something substantial, it's recommended to develop with the same database as you plan on using in production

In addition to a database backend, you'll need to make sure your Python database bindings are installed.

- If you're using PostgreSQL, you'll need the psycopg2 package.
- If you're using MySQL, you'll need a DB API driver like mysqlclient.
- If you're using SQLite you might want to read the SQLite backend notes.
- If you're using Oracle, you'll need a copy of cx\_Oracle

If you plan to use Django's `manage.py migrate` command to automatically create database tables for your models (after first installing Django and creating a project), you'll need to ensure that Django has permission to create and alter tables in the database you're using; if you plan to manually create the tables, you can simply grant Django SELECT, INSERT, UPDATE and DELETE permissions. After creating a database user with these permissions, you'll specify the details in your project's settings file, see DATABASES for details.

Python packages for different databases

Database	Python package	pip installation syntax
PostgreSQL	psycopg2	pip install psycopg2
MySQL	mysql-python	pip install PyMySQL/ pip install mysqlclient
Oracle	cx_Oracle	pip install cx_Oracle
SQLite	Included with the Python distribution	N/A

We need to provide all connection details in the settings file. The settings.py file of our project contains the following code for the database.

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

After providing details, check the connection using the migrate command.

```
python manage.py makemigrations
python manage.py migrate
```

## Starting Project

If this is your first time using Django, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Django project – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, **cd** into a directory where you'd like to store your code, then run the following command:

```
django-admin startproject mysite
```

This will create a **mysite** directory in your current directory.

Let's look at what **startproject** created:

```
mysite/  
  manage.py  
  mysite/  
    __init__.py  
    settings.py  
    urls.py  
    wsgi.py
```

### The development server

Let's verify your Django project works. Change into the outer **mysite** directory, if you haven't already, and run the following commands:

```
python manage.py runserver
```

You'll see the following output on the command line:

```
Performing system checks...  
System check identified no issues (0 silenced).  
You have unapplied migrations; your app may not work properly until they are applied.  
Run 'python manage.py migrate' to apply them.  
August 28, 2019 - 15:50:53  
Django version 1.10, using settings 'mysite.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CONTROL-C.
```

## Django Project Architecture

```
mysite/  
  manage.py  
  mysite/  
    __init__.py  
    settings.py  
    urls.py  
    wsgi.py
```

These files are:

- The outer **mysite/** root directory is just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- **manage.py**: A command-line utility that lets you interact with this Django project in various ways.
- The inner **mysite/** directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. `mysite.urls`).
- **mysite/\_\_init\_\_.py**: An empty file that tells Python that this directory should be considered a Python package.
- **mysite/settings.py**: Settings/configuration for this Django project. Django settings will tell you all about how settings work.
- **mysite/urls.py**: The URL declarations for this Django project; a "table of contents" of your Django-powered site.
- **mysite/wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project.

## Understanding manage.py,

In addition, **manage.py** is automatically created in each Django project. **manage.py** does the same thing as **django-admin** but takes care of a few things for you:

- It puts your project's package on **sys.path**.
- It sets the **DJANGO\_SETTINGS\_MODULE** environment variable so that it points to your project's **settings.py** file.

## Understanding settings.py,

A Django settings file contains all the configuration of your Django installation. A settings file is just a Python module with module-level variables.

Here are a couple of example settings:

```
ALLOWED_HOSTS = ['www.example.com']
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
```

We can manage many other setting of application and it's behavior through **settings.py**.

**Core Settings** : ADMINS, ALLOWED\_HOSTS, CACHES, DATABASES, DATE\_FORMAT, DEBUG, CHARSET, FILE\_UPLOAD, FILE\_STORAGE, EMAIL\_HOST, FORMAT\_MODULE\_PATH, INSTALLED\_APPS, LANGUAGES, LOCALE\_PATHS, MIDDLEWARE, TEMPLATES, TIME\_ZONE, etc.

**Auth**: AUTHENTICATION\_BACKENDS, AUTH\_USER\_MODEL, LOGIN\_REDIRECT\_URL, LOGIN\_URL, LOGOUT\_REDIRECT\_URL, PASSWORD\_HASHERS, AUTH\_PASSWORD\_VALIDATORS

**Messages**: MESSAGE\_LEVEL, MESSAGE\_STORAGE, MESSAGE\_TAGS

**Sessions**: SESSION\_CACHE\_ALIAS, SESSION\_COOKIE\_AGE, SESSION\_COOKIE\_DOMAIN, SESSION\_COOKIE\_HTTPONLY, SESSION\_COOKIE\_NAME, SESSION\_COOKIE\_PATH, SESSION\_COOKIE\_SECURE, SESSION\_ENGINE, SESSION\_SERIALIZER.

**Sites:** SITE\_ID

**Static Files:** STATIC\_ROOT, STATIC\_URL, STATICFILES\_DIRS, STATICFILES\_STORAGE, STATICFILES\_FINDERS

## Understanding \_\_init\_\_.py

The `__init__.py` files are required to make Python treat directories containing the file as packages. This prevents directories with a common name, such as `string`, unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

## Understanding wsgi.py

WSGI is the Web Server Gateway Interface. It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.

Django's primary deployment platform is WSGI, the Python standard for web servers and applications. Django's **startproject** management command sets up a simple default WSGI configuration for you, which you can tweak as needed for your project, and direct any WSGI-compliant application server to use.

## Understanding urls.py and Python regular expression

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations

To design URLs for an app, you create a Python module informally called a **URLconf** (URL configuration). This module is pure Python code and is a simple mapping between URL patterns (simple regular expressions) to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it's pure Python code, it can be constructed dynamically.

### Here's a sample URLconf:

```
from django.conf.urls import url
from . import views
urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

### Notes:

- To capture a value from the URL, just put parenthesis around it.
- There's no need to add a leading slash, because every URL has that. For example, it's `^articles`, not `^/articles`.
- The 'r' in front of each regular expression string is optional but recommended. It tells Python that a string is "raw" – that nothing in the string should be escaped.

**Example requests:**

- A request to `/articles/2005/03/` would match the third entry in the list. Django would call the function `views.month_archive(request, '2005', '03')`.
- `/articles/2005/3/` would not match any URL patterns, because the third entry in the list requires two digits for the month.
- `/articles/2003/` would match the first pattern in the list, not the second one, because the patterns are tested in order, and the first one is the first test to pass. Feel free to exploit the ordering to insert special cases like this. Here, Django would call the function `views.special_case_2003(request)`
- `/articles/2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `/articles/2003/03/03/` would match the final pattern. Django would call the function `views.article_detail(request, '2003', '03', '03')`.

**Understanding admin.py**

`admin.py` consist of all major parts to maintain django-admin panel which comes with django. **`django.contrib.admin`** is package that consists of various other functionality that can be moulded to your application with `admin.py`.

Some of them are **checks, exceptions, templatetags.admin\_urls, utils, views, widgets, etc.** More importantly users can register their models into `admin.py` for CRUD operations of applications.

## Understanding models.py

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses **`django.db.models.Model`**.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API

**Example:**

This example model defines a `Person`, which has a `first_name` and `last_name`:  
from `django.db import models`.

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` and `last_name` are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column. The above `Person` model would



create a database table like this:

```
CREATE TABLE myapp_person (  
    "id" serial NOT NULL PRIMARY KEY,  
    "first_name" varchar(30) NOT NULL,  
    "last_name" varchar(30) NOT NULL  
);
```

Some technical notes:

- The name of the table, myapp\_person, is automatically derived from some model metadata but can be overridden. See Table names for more details.
- An id field is added automatically, but this behavior can be overridden. See Automatic primary key fields.
- The CREATE TABLE SQL in this example is formatted using PostgreSQL syntax, but it's worth noting Django uses SQL tailored to the database backend specified in your settings file.

## Understanding views.py

### **class django.views.generic.base.View**

The master class-based base view. All other class-based views inherit from this base class. It isn't strictly a generic view and thus can also be imported from django.views.

Method Flowchart

- setup()
- dispatch()
- http\_method\_not\_allowed()
- options()

### **Example views.py:**

```
from django.http import HttpResponse  
from django.views import View  
  
class MyView(View):  
  
    def get(self, request, *args, **kwargs):  
        return HttpResponse('Hello, World!')
```

### **Example urls.py:**

```
from django.urls import path  
  
from myapp.views import MyView  
  
urlpatterns = [  
    path('mine/', MyView.as_view(), name='my-view'),  
]
```

## Template system basics

Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships built-in backends for its own template system, creatively called the Django template language (DTL), and for the popular alternative Jinja2. Backends for other template languages may be available from third-parties.

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

The Django template language is Django's own template system. Until Django 1.8 it was the only built-in option available. It's a good template library even though it's fairly opinionated and sports a few idiosyncrasies. If you don't have a pressing reason to choose another backend, you should use the DTL, especially if you're writing a pluggable application and you intend to distribute templates. Django's contrib apps that include templates, like `django.contrib.admin`, use the DTL.

For historical reasons, both the generic support for template engines and the implementation of the Django template language live in the `django.template` namespace.

## Using template system

A Django template is simply a text document or a Python string marked-up using the Django template language. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags.

A template is rendered with a context. Rendering replaces variables with their values, which are looked up in the context, and executes tags. Everything else is output as is. The syntax of the Django template language involves four constructs.

### 1. Variables :

A variable outputs a value from the context, which is a dict-like object mapping keys to values. Variables are surrounded by `{{` and `}}` like this:

### 2. Tags :

Tags provide arbitrary logic in the rendering process. This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an "if" statement or a "for" loop, grab content from a database, or even enable access to other template tags. Tags are surrounded by `{%` and `%}`

### 3. Filters :

Filters transform the values of variables and tag arguments.

```
{{ django|title }}
```

### 4. Comments :

`{# this won't be rendered #}`, A `{% comment %}` tag provides multi-line comments.

## Basic Template Tags and Filters

### autoescape

Controls the current auto-escaping behavior. This tag takes either on or off as an argument and that determines whether auto-escaping is in effect inside the block. The block is closed with an endautoescape ending tag.

When auto-escaping is in effect, all variable content has HTML escaping applied to it before placing the result into the output (but after any filters have been applied). This is equivalent to manually applying the escape filter to each variable.

```
{% autoescape on %}
  {{ body }}
{% endautoescape %}
```

### block

Defines a block that can be overridden by child templates.

### comment

Ignores everything between {% comment %} and {% endcomment %}. An optional note may be inserted in the first tag. For example, this is useful when commenting out code for documenting why the code was disabled. Comment tags cannot be nested.

### csrf\_token

This tag is used for CSRF protection. For more information on Cross Site Request Forgeries (**CSRF**).

### cycle

Produces one of its arguments each time this tag is encountered. The first argument is produced on the first encounter, the second argument on the second encounter, and so forth. Once all arguments are exhausted, the tag cycles to the first argument and produces it again. This tag is particularly useful in a loop:

```
{% for o in some_list %}
  <tr class="{% cycle 'row1' 'row2' %}">
    ...
  </tr>
{% endfor %}
```

The first iteration produces HTML that refers to class row1, the second to row2, the third to row1 again, and so on for each iteration of the loop. You can use variables, too. For example, if you have two template variables, rowvalue1 and rowvalue2, you can alternate between their values like this:

```
{% for o in some_list %}
  <tr class="{% cycle rowvalue1 rowvalue2 %}">
    ...
  </tr>
{% endfor %}
```

You can use any number of values in a cycle tag, separated by spaces. Values enclosed in single quotes (') or double quotes (") are treated as string literals, while values without quotes are treated as template variables.

**debug**

Outputs a whole load of debugging information, including the current context and imported modules.

**extends**

Signals that this template extends a parent template. This tag can be used in two ways:

1. `{% extends "base.html" %}` (with quotes) uses the literal value "base.html" as the name of the parent template to extend.
2. `{% extends variable %}` uses the value of variable. If the variable evaluates to a string, Django will use that string as the name of the parent template. If the variable evaluates to a Template object, Django will use that object as the parent template.

**filter**

Filters the contents of the block through one or more filters.

**firstof**

Outputs the first argument variable that is not False. Outputs nothing if all the passed variables are False.

Sample usage:

```
{% firstof var1 var2 var3 %}
```

This is equivalent to:

```
{% if var1 %}
```

```
  {{ var1 }}
```

```
{% elif var2 %}
```

```
  {{ var2 }}
```

```
{% elif var3 %}
```

```
  {{ var3 }}
```

```
{% endif %}
```

**for**

Loops over each item in an array, making the item available in a context variable. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
```

```
{% for athlete in athlete_list %}
```

```
  <li>{{ athlete.name }}</li>
```

```
{% endfor %}
```

```
</ul>
```

You can loop over a list in reverse by using `{% for obj in list reversed %}`. If you need to loop over a list of lists, you can unpack the values in each sub list into individual variables. This can also be useful if you need to access the items in a dictionary. For example, if your context contained a dictionary data, the following would display the keys and values of the dictionary:

```
{% for key, value in data.items %}
```

```
  {{ key }}: {{ value }}
```

```
{% endfor %}
```

**for ... empty**

The for tag can take an optional `{% empty %}` clause whose text is displayed if the given array is empty or could not be found:

```
<ul>
{% for athlete in athlete_list %}
  <li>{{ athlete.name }}</li>
{% empty %}
  <li>Sorry, no athletes in this list.</li>
{% endfor %}
</ul>
```

**if**

The `{% if %}` tag evaluates a variable, and if that variable is true (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output:

```
{% if athlete_list %}
  Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
  Athletes should be out of the locker room soon!
{% else %}
  No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable. As you can see, the if tag may take one or several `{% elif %}` clauses, as well as an `{% else %}` clause that will be displayed if all previous conditions fail. These clauses are optional.

**Boolean operators**

if tags may use `and`, `or` or `not` to test a number of variables or to negate a given variable:

```
{% if athlete_list and coach_list %}
  Both athletes and coaches are available.
{% endif %}
```

```
{% if not athlete_list %}
  There are no athletes.
{% endif %}
```

```
{% if athlete_list or coach_list %}
  There are some athletes or some coaches.
{% endif %}
```

Use of both `and` and `or` clauses within the same tag is allowed, with `and` having higher precedence than `or` e.g.:

```
{% if athlete_list and coach_list or cheerleader_list %}
  will be interpreted like:
if (athlete_list and coach_list) or cheerleader_list
```

Use of actual parentheses in the if tag is invalid syntax. If you need them to indicate precedence, you should use nested if tags.

if tags may also use the operators `==`, `!=`, `<`, `>`, `<=`, `>=` and in which work as listed in Table E-1.

### Complex expressions

All of the above can be combined to form complex expressions. For such expressions, it can be important to know how the operators are grouped when the expression is evaluated – that is, the precedence rules. The precedence of the operators, from lowest to highest, is as follows:

- or
- and
- not
- in
- ==, !=, <, >, <=, >=

This order of precedence follows Python exactly.

### Filters

You can also use filters in the if expression. For example:

```
{% if messages|length >= 100 %}
You have lots of messages today!
{% endif %}
```

### ifchanged

Check if a value has changed from the last iteration of a loop. The `{% ifchanged %}` block tag is used within a loop. It has two possible uses.

1. Checks its own rendered contents against its previous state and only displays the content if it has changed.
2. If given one or more variables, check whether any variable has changed.

### ifequal

Output the contents of the block if the two arguments equal each other. Example:

```
{% ifequal user.pk comment.user_id %}
...
{% endifequal %}
```

An alternative to the `ifequal` tag is to use the `if` tag and the `==` operator.

### ifnotequal

Just like `ifequal`, except it tests that the two arguments are not equal. An alternative to the `ifnotequal` tag is to use the `if` tag and the `!=` operator.

### include

Loads a template and renders it with the current context. This is a way of including other templates within a template. The template name can either be a variable:

```
{% include template_name %}
or a hard-coded (quoted) string:
{% include "foo/bar.html" %}
```

### load

Loads a custom template tag set. For example, the following template would load all the tags and filters registered in `somelibrary` and `otherlibrary` located in package `package`:

```
{% load somelibrary package.otherlibrary %}
```



You can also selectively load individual filters or tags from a library, using the `from` argument.

In this example, the template tags/filters named `foo` and `bar` will be loaded from `somelibrary`:

```
{% load foo bar from somelibrary %}
```

### lorem

Displays random lorem ipsum Latin text. This is useful for providing sample data in templates. Usage:

```
{% lorem [count] [method] [random] %}
```

The `{% lorem %}` tag can be used with zero, one, two or three arguments. The arguments are:

1. **Count.** A number (or variable) containing the number of paragraphs or words to generate (default is 1).
2. **Method.** Either `w` for words, `p` for HTML paragraphs or `b` for plain-text paragraph blocks (default is `b`).
3. **Random.** The word `random`, which if given, does not use the common paragraph (Lorem ipsum dolor sit amet...) when generating text.

For example, `{% lorem 2 w random %}` will output two random Latin words.

### now

Displays the current date and/or time, using a format according to the given string. Such string can contain format specifiers characters as described in the date filter section.

Example:

```
It is {% now "jS F Y H:i" %}
```

The format passed can also be one of the predefined ones `DATE_FORMAT`, `DATETIME_FORMAT`, `SHORT_DATE_FORMAT` or `SHORT_DATETIME_FORMAT`. The predefined formats may vary depending on the current locale and if format-localization is enabled, e.g.:

```
It is {% now "SHORT_DATETIME_FORMAT" %}
```

### spaceless

Removes whitespace between HTML tags. This includes tab characters and newlines.

Example usage:

```
{% spaceless %}
<p>
  <a href="foo/">Foo</a>
</p>
{% endspaceless %}
```

This example would return this HTML:

```
<p><a href="foo/">Foo</a></p>
```

### with

Caches a complex variable under a simpler name. This is useful when accessing an expensive method (e.g., one that hits the database) multiple times. For example:

```
{% with total=business.employees.count %}
  {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

## Built-in filters

**add** : Adds the argument to the value. For example:

```
{{ value|add:"2" }}
```

If value is 4, then the output will be 6.

**addslashes** : Adds slashes before quotes. Useful for escaping strings in CSV, for example.

```
{{ value|addslashes }}
```

If value is "I'm using Django", the output will be "I\'m using Django".

**capfirst** : Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.

**center** : Centers the value in a field of a given width.

```
"{{ value|center:"14" }}"
```

If value is "Django", the output will be " Django "

**cut** : Removes all values of arg from the given string.

**date** : Formats a date according to the given format. Uses a similar format as PHP's date() function with some differences. These format characters are not used in Django outside of templates. They were designed to be compatible with PHP to ease transitioning for designers.

```
{{ value|date:"D d M Y" }}
```

If value is a datetime object (e.g., the result of datetime.datetime.now()), the output will be the string "Fri 01 Jul 2016". The format passed can be one of the predefined ones DATE\_FORMAT, DATETIME\_FORMAT, SHORT\_DATE\_FORMAT or SHORT\_DATETIME\_FORMAT, or a custom format that uses date format specifiers.

**default**: If value evaluates to False, uses the given default. Otherwise, uses the value.

```
{{ value|default:"nothing" }}
```

**default\_if\_none** : If (and only if) value is None, uses the given default. Otherwise, uses the value.

**dictsort** : Takes a list of dictionaries and returns that list sorted by the key given in the argument.

```
{{ value|dictsort:"name" }}
```

**divisibleby**: Returns True if the value is divisible by the argument. For example:

```
{{ value|divisibleby:"3" }}
```

If value is 21, the output would be True.

**escape** : Escapes a string's HTML. Specifically, it makes these replacements:

- < is converted to &lt;
- > is converted to &gt;
- ' (single quote) is converted to &#39;
- " (double quote) is converted to &quot;
- & is converted to &amp;

The escaping is only applied when the string is output, so it does not matter where in a chained sequence of filters you put escape: it will always be applied as though it were the last filter.

**escapejs**: Escapes characters for use in JavaScript strings. This does *not* make the string safe for use in HTML, but does protect you from syntax errors when using templates to generate JavaScript/JSON.

**filesizeformat**: Formats the value like a 'human-readable' file size (i.e. '13 KB', '4.1 MB', '102 bytes', etc.).

```
{{ value|filesizeformat }}
```

If value is "123456789", the output would be 117.7 MB.

**first** : Returns the first item in a list.

**floatformat** : When used without an argument, rounds a floating-point number to one decimal place – but only if there's a decimal part to be displayed. If used with a numeric integer argument, floatformat rounds a number to that many decimal places.

```
For example, if value is 34.23234, {{ value|floatformat:3 }} will output 34.232.
```

**get\_digit**: Given a whole number, returns the requested digit, where 1 is the right-most digit.

**join**: Joins a list with a string, like Python's str.join(list).

**last**: Returns the last item in a list.

**length**: Returns the length of the value. This works for both strings and lists.

**length\_is**: Returns True if the value's length is the argument, or False otherwise. For example:

```
{{ value|length_is:"4" }}
```

**linebreaks**: Replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break (<br />) and a new line followed by a blank line becomes a paragraph break (</p>).

**linebreaksbr** : Converts all newlines in a piece of plain text to HTML line breaks (<br />).

**linenumbers**: Displays text with line numbers.

**ljust:** Left-aligns the value in a field of a given width. For example:

```
{{ value|ljust:"10" }}
```

If value is "Django", the output will be "Django".

**lower :** Converts a string into all lowercase.

**make\_list :** Returns the value turned into a list. For a string, it's a list of characters. For an integer, the argument is cast into an Unicode string before creating a list.

**phone2numeric:** Converts a phone number (possibly containing letters) to its numerical equivalent. The input doesn't have to be a valid phone number. This will happily convert any string. For example:

```
{{ value|phone2numeric }}
```

If value is 800-COLLECT, the output will be 800-2655328.

**pluralize:** Returns a plural suffix if the value is not 1. By default, this suffix is "s". For words that don't pluralize by simple suffix, you can specify both a singular and plural suffix, separated by a comma. Example:

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

**random:** Returns a random item from the given list.

**rjust:** Right-aligns the value in a field of a given width. For example:

```
{{ value|rjust:"10" }}
```

If value is "Django", the output will be " Django".

**slice:** Returns a slice of the list. Uses the same syntax as Python's list slicing.

**slugify:** Converts to ASCII. Converts spaces to hyphens. Removes characters that aren't alphanumeric, underscores, or hyphens. Converts to lowercase. Also strips leading and trailing whitespace.

**stringformat:** Formats the variable according to the argument, a string formatting specifier. This specifier uses Python string formatting syntax, with the exception that the leading % is dropped.

**time:** Formats a time according to the given format. Given format can be the predefined one TIME\_FORMAT, or a custom format, same as the date filter.

**timesince:** Formats a date as the time since that date (e.g., 4 days, 6 hours). Takes an optional argument that is a variable containing the date to use as the comparison point (without the argument, the comparison point is now).

**timeuntil:** Measures the time from now until the given date or datetime.

**title:** Converts a string into title case by making words start with an uppercase character and the remaining characters lowercase.

**truncatechars:** Truncates a string if it is longer than the specified number of characters. Truncated strings will end with a translatable ellipsis sequence (...). For example:

```
{{ value|truncatechars:9 }}
```

**truncatechars\_html:** Similar to truncatechars, except that it is aware of HTML tags.

**truncatewords:** Truncates a string after a certain number of words.

**truncatewords\_html:** Similar to truncatewords, except that it is aware of HTML tags.

**unordered\_list:** Recursively takes a self-nested list and returns an HTML unordered list – without opening and closing tags.

**upper:** Converts a string into all uppercase.

**urlize:** Converts URLs and email addresses in text into clickable links. This template tag works on links prefixed with `http://`, `https://`, or `www..`

**urlizetrunc:** Converts URLs and email addresses into clickable links just like urlize, but truncates URLs longer than the given character limit. For example:

```
{{ value|urlizetrunc:15 }}
```

If value is “Check out [www.djangoproject.com](http://www.djangoproject.com)”, the output would be “Check out <a href="http://www.djangoproject.com" rel="nofollow">www.djangopr...</a>”. As with urlize, this filter should only be applied to plain text.

**wordcount:** Returns the number of words.

**wordwrap :** Wraps words at specified line length.

**yesno:** Maps values for true, false and (optionally) None, to the strings yes, no, maybe, or a custom mapping passed as a comma-separated list, and returns one of those strings according to the value: For example:

```
{{ value|yesno:"yeah,no,maybe" }}
```

## Internationalization Tags and Filters

Django provides template tags and filters to control each aspect of internationalization in templates. They allow for granular control of translations, formatting, and time zone conversions.

**i18n:** This library allows specifying translatable text in templates. To enable it, set `USE_I18N` to `True`, then load it with `{% load i18n %}`.

**l10n:** This library provides control over the localization of values in templates. You only need to load the library using `{% load l10n %}`, but you'll often set `USE_L10N` to `True` so that localization is active by default.

**tz:** This library provides control over time zone conversions in templates. Like `l10n`, you only need to load the library using `{% load tz %}`, but you'll usually also set `USE_TZ` to `True` so that conversion to local time happens by default. See `time-zones-in-templates`.

## Other Tags and Filters Libraries

**static :** To link to static files that are saved in `STATIC_ROOT` Django ships with a static template tag. You can use this regardless if you're using `RequestContext` or not.

```
{% load static %}

```

It is also able to consume standard context variables, e.g. assuming `user_stylesheet` variable is passed to the template:

```
{% load static %}
<link rel="stylesheet" href="{% static user_stylesheet %}" type="text/css" media="screen" />
```

If you'd like to retrieve a static URL without displaying it, you can use a slightly different call.



## Using Templates in Views

A template is a text document, or a normal Python string, that is marked up using the Django template language. A template can contain block tags and variables.

A block tag is a symbol within a template that does something. This definition is deliberately vague. For example, a block tag can produce content, serve as a control structure (an if statement or for loop), grab content from a database, or enable access to other template tags.

Block tags are surrounded by `{% and %}`:

```
{% if is_logged_in %}
    Thanks for logging in!
{% else %}
    Please log in.
{% endif %}
```

A variable is a symbol within a template that outputs a value. Variable tags are surrounded by `{{ and }}`:

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

A context is a name-value mapping (similar to a Python dictionary) that is passed to a template.

A template renders a context by replacing the variable “holes” with values from the context and executing all block tags.

Below example shows that Template is integrated into index.html.

```
#index.html
{% extends "main.html" %}
{% block content %}
<h1>{{ title }}</h1>
<div class="container">
</UL>
</div>
{% endblock content %}
```

```
#views.py
from django.shortcuts import render
def index(request):
    context = {
        'title': 'This is Index Page.',
    }
    return render(request, 'index.html', context)
```

`render()` is a special Django helper function that creates a shortcut for communicating with a web browser.

You can code each of these steps separately in Django, but in the vast majority of cases it's more common (and easier) to use Django's **render()** function, which provides a shortcut that provides all three steps in a single function.

When you supply the original request, the template and a context directly to **render()**, it returns the appropriately formatted response without you having to code the intermediate steps.

In our modified `views.py`, we are returning the original request object from the browser, the name of our site template and a dictionary (the context) containing our title and `cal` variables from the view.

Once you have modified your `views.py` file, save it and fire up the development server. If you navigate to **<http://127.0.0.1:8000/index/>**



## Template loading

Generally, you'll store templates in files on your filesystem, but you can use custom template loaders to load templates from other sources.

Django has two ways to load templates:

- **django.template.loader.get\_template(template\_name):** `get_template` returns the compiled template (a `Template` object) for the template with the given name. If the template doesn't exist, a `TemplateDoesNotExist` exception will be raised.
- **django.template.loader.select\_template(template\_name\_list):** `select_template` is just like `get_template`, except it takes a list of template names. Of the list, it returns the first template that exists. If none of the templates exist, a `TemplateDoesNotExist` exception will be raised.

Each of these functions by default uses your **TEMPLATE\_DIRS** setting to load templates. Internally, however, these functions actually delegate to a template loader for the heavy lifting.

Some of loaders are disabled by default, but you can activate them by editing the **TEMPLATE\_LOADERS** setting. **TEMPLATE\_LOADERS** should be a tuple of strings, where each string represents a template loader. These template loaders ship with Django:

- `django.template.loaders.filesystem.load_template_source`: This loader loads templates from the filesystem, according to **TEMPLATE\_DIRS**. It is enabled by default.
- `django.template.loaders.app_directories.load_template_source`: This loader loads templates from Django applications on the filesystem. For each application in `INSTALLED_APPS`, the loader looks for a `templates` subdirectory. If the directory exists, Django looks for templates there.

This means you can store templates with your individual applications, making it easy to distribute Django applications with default templates. For example, if `INSTALLED_APPS` contains ('myproject.polls', 'myproject.music'), then `get_template('foo.html')` will look for templates in this order:

- `/path/to/myproject/polls/templates/foo.html`
- `/path/to/myproject/music/templates/foo.html`

Note that the loader performs an optimization when it is first imported: it caches a list of the `INSTALLED_APPS` packages that have a `templates` subdirectory. This loader is enabled by default.

- `django.template.loaders.eggs.load_template_source`: This loader is just like `app_directories`, except it loads templates from Python eggs rather than from the filesystem. This loader is disabled by default; you'll need to enable it if you're using eggs to distribute your application.

Django uses the template loaders in order according to the **TEMPLATE\_LOADERS** setting. It uses each loader until a loader finds a match

## Configuring database

Open up **mysite/settings.py**. It's a normal Python module with module-level variables representing Django settings.

By default, the configuration uses SQLite. If you're new to databases, or you're just interested in trying Django, this is the easiest choice. SQLite is included in Python, so you won't need to install anything else to support your database. When starting your first real project, however, you may want to use a more scalable database like PostgreSQL, to avoid database-switching headaches down the road.

- **ENGINE** – Either `'django.db.backends.sqlite3'`, `'django.db.backends.postgresql'`, `'django.db.backends.mysql'`, or `'django.db.backends.oracle'`. Other backends are also available.
- **NAME** – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, **NAME** should be the full absolute path, including filename, of that file. The default value, `os.path.join(BASE_DIR, 'db.sqlite3')`, will store the file in your project directory.

While you're editing **mysite/settings.py**, set **TIME\_ZONE** to your time zone. Also, note the **INSTALLED\_APPS** setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects.

By default, **INSTALLED\_APPS** contains the following apps, all of which come with Django:

- **django.contrib.admin** – The admin site. You'll use it shortly.
- **django.contrib.auth** – An authentication system.
- **django.contrib.contenttypes** – A framework for content types.
- **django.contrib.sessions** – A session framework.
- **django.contrib.messages** – A messaging framework.
- **django.contrib.staticfiles** – A framework for managing static files.

These applications are included by default as a convenience for the common case. Some of these applications make use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

```
python manage.py migrate
```

Operations to perform:

Apply all migrations: admin, auth, contenttypes, sessions

**Running migrations:**

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

Applying admin.0001\_initial... OK

Applying admin.0002\_logentry\_remove\_auto\_add... OK

```
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying sessions.0001_initial... OK
```

The **migrate** command looks at the **INSTALLED\_APPS** setting and creates any necessary database tables according to the database settings in your **mysite/settings.py** file and the database migrations shipped with the app (we'll cover those later). You'll see a message for each migration it applies.



## Defining models

Now we'll define your models – essentially, your database layout, with additional metadata. In our simple poll app, we'll create two models: **Question** and **Choice**. A **Question** has a question and a publication date. A **Choice** has two fields: the text of the choice and a vote tally. Each **Choice** is associated with a **Question**.

These concepts are represented by simple Python classes. Edit the `polls/models.py` file so it looks like this:

```
# polls/models.py
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

The code is straightforward. Each model is represented by a class that subclasses `django.db.models.Model`. Each model has a number of class variables, each of which represents a database field in the model.

Each field is represented by an instance of a **Field** class – e.g., **CharField** for character fields and **DateTimeField** for datetimes. This tells Django what type of data each field holds. The name of each **Field** instance (e.g. `question_text` or `pub_date`) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name.

You can use an optional first positional argument to a **Field** to designate a human-readable name. That's used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn't provided, Django will use the machine-readable name. In this example, we've only defined a human-readable name for **Question.pub\_date**. For all other fields in this model, the field's machine-readable name will suffice as its human-readable name.

Some **Field** classes have required arguments. **CharField**, for example, requires that you give it a `max_length`. That's used not only in the database schema, but in validation, as we'll soon see.

A **Field** can also have various optional arguments; in this case, we've set the `default` value of `votes` to 0.

Finally, note a relationship is defined, using **ForeignKey**. That tells Django each **Choice** is related to a single **Question**. Django supports all the common database relationships: many-to-one, many-to-many, and one-to-one.



Let's run another command:

```
python manage.py makemigrations
```

You should see something similar to the following:

Migrations for 'polls':

polls/migrations/0001\_initial.py:

- Create model Choice
- Create model Question
- Add field question to choice

By running **makemigrations**, you're telling Django that you've made some changes to your models (in this case, you've made new ones) and that you'd like the changes to be stored as a *migration*.

Migrations are how Django stores changes to your models (and thus your database schema) - they're just files on disk. You can read the migration for your new model if you like; it's the file **polls/migrations/0001\_initial.py**. Don't worry, you're not expected to read them every time Django makes one, but they're designed to be human-editable in case you want to manually tweak how Django changes things.

Now, run **migrate** again to create those model tables in your database:

```
python manage.py migrate
```

Operations to perform:

Apply all migrations: admin, auth, contenttypes, **polls**, sessions

Running migrations:

Rendering model states... DONE

Applying polls.0001\_initial... OK

The **migrate** command takes all the migrations that haven't been applied (Django tracks which ones are applied using a special table in your database called **django\_migrations**) and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database.

Migrations are very powerful and let you change your models over time, as you develop your project, without the need to delete your database or tables and make new ones - it specializes in upgrading your database live, without losing data. We'll cover them in more depth in a later part of the tutorial, but for now, remember the three-step guide to making model changes:

- Change your models (in **models.py**).
- Run **python manage.py makemigrations** to create migrations for those changes
- Run **python manage.py migrate** to apply those changes to the database.

The reason that there are separate commands to make and apply migrations is because you'll commit migrations to your version control system and ship them with your app; they not only make your development easier, they're also useable by other developers and in production.

## Basic data access

Now, let's hop into the interactive Python shell and play around with the free API Django gives you. To invoke the Python shell, use this command:

```
python manage.py shell
```

We're using this instead of simply typing "python", because **manage.py** sets the **DJANGO\_SETTINGS\_MODULE** environment variable, which gives Django the Python import path to your **mysite/settings.py** file.

### Inserting /Creating Objects

You get a `QuerySet` by using your model's `Manager`. Each model has at least one `Manager`, and it's called `objects` by default.

```
>>> from polls.models import Question
>>> qi = Question(question_text='What is your name?',pub_date='2019-08-08 00:00:00')
>>> qi.save()
```

This performs an **INSERT** SQL statement behind the scenes. Django doesn't hit the database until you explicitly call `save()`. The `save()` method has no return value.

### Selecting objects

To retrieve objects from your database, construct a `QuerySet` via a `Manager` on your model class.

A **QuerySet** represents a collection of objects from your database. It can have zero, one or many *filters*. Filters narrow down the query results based on the given parameters. In SQL terms, a `QuerySet` equates to a `SELECT` statement, and a filter is a limiting clause such as `WHERE` or `LIMIT`.

You get a `QuerySet` by using your model's `Manager`. Each model has at least one `Manager`, and it's called `objects` by default.

The **Manager** is the main source of **QuerySets** for a model. For example, **Blog.objects.all()** returns a **QuerySet** that contains all **Blog** objects in the database.

The simplest way to retrieve objects from a table is to get all of them. To do this, use the **all()** method on a **Manager**:

```
>>>Question.objects.all()
QuerySet [<Question: Question object>, <Question: Question object>]
```

If you have seen that in above output we are having multiple Question Objects that are not showing which record are inserted.

To represent them properly in model.py inside Questions class define `__str__` function to describe the returned result when you query the objects.

```
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

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

Now run the below command to check again.

```
>>>Question.objects.all()
<QuerySet [<Question: What is your name?>, <Question: What is your name?>]>
```

From above output you can that objects have been replaced with Name of the question and also we can see one problem that we have inserted multiple records. To solve such problem we will update the records by retrieving with ids.

### Retrieving a single object with get()

If you know there is only one object that matches your query, you can use the `get()` method on a **Manager** which returns the object directly:

```
>>> Question.objects.get(id=1)
<Question: What is your name?>

>>> Question.objects.get(id=2)
<Question: What is your name?>
```

### Updating Objects

To save changes to an object that's already in the database, we will get object with id inside "qu" variable to retrieve the whole record and use `save()` to make changes in that table of database.

```
>>> qu = Question.objects.get(id=2)
>>> qu.question_text='What is Your Age?'
>>> qu.save()

>>> Question.objects.get(id=2)
<Question: What is Your Age?>

>>> Question.objects.all()
<QuerySet [<Question: What is your name?>, <Question: What is Your Age?>]>
```

In above output updation is successful.

## Deleting objects

The delete method, conveniently, is named `delete()`. This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type.

To delete record let us create/insert a new record inside Question table.

```
>>> qi = Question(question_text='Where do you live?',pub_date='2019-08-08 00:00:00')
>>> qi.save()
```

Let us see the new record is inserted inside table.

```
>>> Question.objects.all()
<QuerySet [<Question: What is your name?>, <Question: What is Your Age?>, <Question: Where do you live?>, <Question: Where do you live?>]>
```

Let us delete the inserted record.

```
>>> qd = Question.objects.get(id=3)
>>> qd.delete()
(1, {'testApp.Choice': 0, 'testApp.Question': 1})
```

Let us check that objects is deleted or not.

```
>>> Question.objects.get(id=3)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\STUD\Env\VirtualDjango\lib\site-packages\django\db\models\manager.py", line 85, in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "C:\Users\STUD\Env\VirtualDjango\lib\site-packages\django\db\models\query.py", line 385, in get
    self.model._meta.object_name
testApp.models.DoesNotExist: Question matching query does not exist.
```

## Activating the Admin interface

One of the most powerful parts of Django is the automatic admin interface. It reads metadata from your models to provide a quick, model-centric interface where trusted users can manage content on your site. The admin's recommended use is limited to an organization's internal management tool. It's not intended for building your entire front end around.

The admin has many hooks for customization, but beware of trying to use those hooks exclusively. If you need to provide a more process-centric interface that abstracts away the implementation details of database tables and fields, then it's probably time to write your own views.

### Overview

The admin is enabled in the default project template used by **startproject**.

For reference, here are the requirements:

1. Add **'django.contrib.admin'** to your **INSTALLED\_APPS** setting.
2. The admin has four dependencies - **django.contrib.auth**, **django.contrib.contenttypes**, **django.contrib.messages** and **django.contrib.sessions**. If these applications are not in your **INSTALLED\_APPS** list, add them.

```
# Application definition
= [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog1',
]
```

3. Add **django.contrib.auth.context\_processors.auth** and **django.contrib.messages.context\_processors.messages** to the **'context\_processors'** option of the **DjangoTemplates** backend defined in your **TEMPLATES** as well as **django.contrib.auth.middleware.AuthenticationMiddleware** and **django.contrib.messages.middleware.MessageMiddleware** to **MIDDLEWARE**. These are all active by default, so you only need to do this if you've manually tweaked the settings.

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

4. Determine which of your application's models should be editable in the admin interface.
5. For each of those models, optionally create a **ModelAdmin** class that encapsulates the customized admin functionality and options for that particular model.

6. Instantiate an **AdminSite** and tell it about each of your models and **ModelAdmin** classes.

```
from django.contrib import admin
from .models import studInfo
# Register your models here.
admin.site.register(studInfo)
```

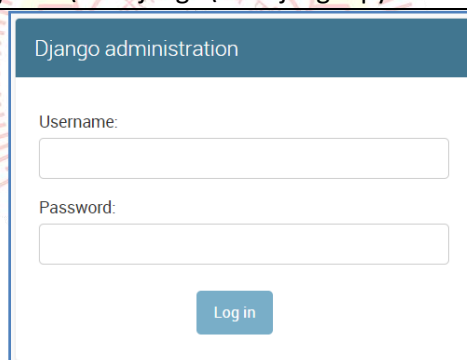
7. Hook the **AdminSite** instance into your URLconf.

```
from django.contrib import admin
urlpatterns = [
    url(r'^admin/', admin.site.urls)
]
```

8. If you need to create a user to login with, you can use the **createsuperuser** command.

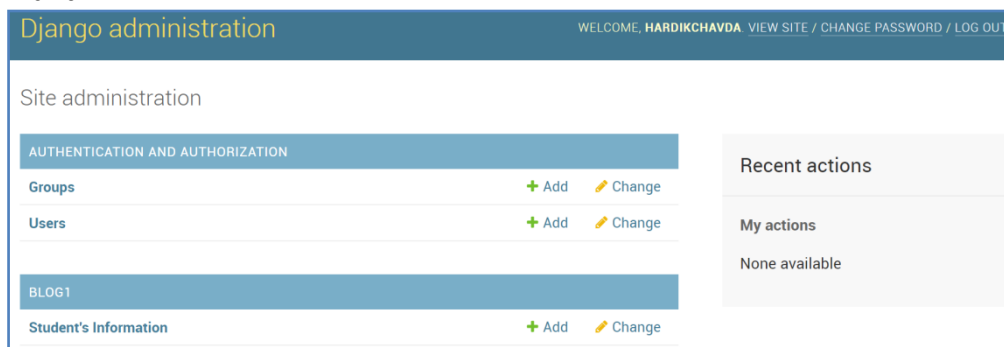
```
(HELLOW~1) D:\Python\testDjango\testDjango>python manage.py createsuperuser
{'Telegram', 'Postcard', 'Radio'}
Username (leave blank to use 'hardikchavda'):
Email address: hardikkchavda@gmail.com
Password:
Password (again):
Superuser created successfully.
```

```
(HELLOW~1) D:\Python\testDjango\testDjango>python manage.py runserver
```



9. Then open your browser and open adminpanel by writing <http://127.0.0.1:8000/admin> and you will see the login screen page.

10. Login Using the username and password you created with **createsuperuser** command.



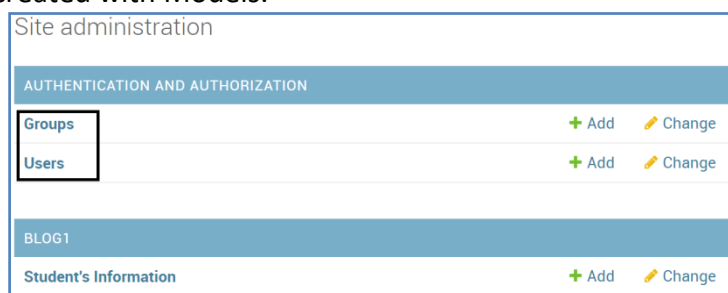
11. That's it you've successfully created the admin panel for your application.



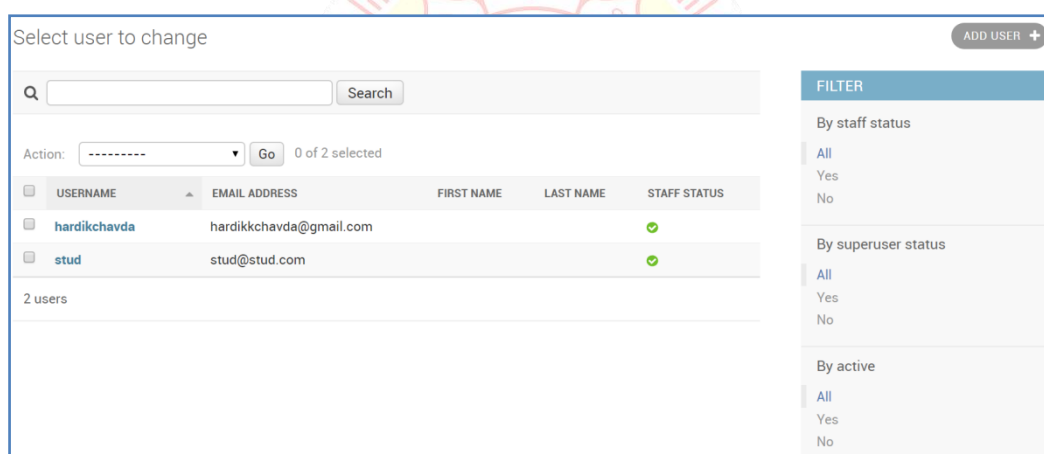
## Using the Admin site

Django admin site administration panel is useful for all types of CRUD operations with selected database. By Default after creating SuperUser you will be able to see Authentication and Authorization Panel which administers Users and their roles for Database Tables.

Two Database Default objects Groups and Users will be on TOP, which manages Roles for all the tables created with Models.



Groups will be empty by default and Users will have all the users you have created with superuser command.



User panel lets you show all the users listed and with their details you can search users by using the searchbar.

On the right side panel you can filter the Users by their Status.

The screenshot shows the Django Admin interface for editing a user. At the top, it says 'Django administration' and 'WELCOME, HARDIKCHAVDA'. The breadcrumb trail is 'Home > Authentication and Authorization > Users > hardikchavda'. The main heading is 'Change user' with a 'HISTORY' button. The 'Username' field is 'hardikchavda'. The 'Password' field shows the algorithm 'pbkdf2\_sha256', iterations '30000', salt '8xcYz8\*\*\*\*\*', and hash 'rUNA4J\*\*\*\*\*'. Below this is the 'Personal info' section with fields for 'First name', 'Last name', and 'Email address' (hardikkchavda@gmail.com). The 'Permissions' section has three checked checkboxes: 'Active', 'Staff status', and 'Superuser status'. There are two selection panels: 'Groups' and 'User permissions', each with an 'Available' list and a 'Chosen' list. The 'Important dates' section shows 'Last login' on 2019-09-08 at 10:11:26 and 'Date joined' on 2019-09-06 at 22:51:48. At the bottom are buttons for 'Delete', 'Save and add another', 'Save and continue editing', and 'SAVE'.

On entering the user page you can manage a users **Personal Info, Permissions and Important Dates.**

## Adding Records

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#) [Change](#)

Users [+ Add](#) [Change](#)

---

BLOG1

Student's Information [+ Add](#) [Change](#)

All the other tables created by user through models will be listed below Authentication and Authorization panel. By clicking on **Add** user can insert records inside the table.

Add Student Information

Student Name:

Address:

City:

Contact Number:

[Save and add another](#) [Save and continue editing](#) [SAVE](#)

A user can enter records here and can save the details, on clicking Save and add another the form will be insert the records and reset the form for another record to save.

On Clicking Save and continue editing Record will be inserted and form will be filled with values inserted. On Clicking Save record will be inserted and you will redirect to previous page.

Add Student Information

Please correct the errors below.

Student Name:  This field is required.

Address:  This field is required.

City:

Contact Number:  Enter a whole number.

[Save and add another](#) [Save and continue editing](#) [SAVE](#)

**Note:** Remember django forms will inherit validations from models such as (null=True, blank=True) etc.

## Updating and Deleting Records

Select Student Information to change ADD STUDENT INFORMATION+

Action: -----  0 of 3 selected

<input type="checkbox"/>	STUDENT INFORMATION
<input type="checkbox"/>	Mr. India
<input type="checkbox"/>	Missile Man
<input type="checkbox"/>	Captain India

3 Student's Information

On Clicking on change button django admin shows you list of available records inside the table and by clicking on them records are fetched from the database to either delete or update the record.

Change Student Information HISTORY

Student Name:

Address:

City:

Contact Number:

You will see Delete Button and Save Buttons which you can use make changes to current record.

Select Student Information to change ADD STUDENT INFORMATION+

Action: -----  2 of 3 selected

Delete selected Student's Information

<input type="checkbox"/>	STUDENT INFORMATION
<input checked="" type="checkbox"/>	Mr. India
<input checked="" type="checkbox"/>	Missile Man
<input type="checkbox"/>	Captain India

3 Student's Information

You can also delete multiple records from the tables list view if required, remember when you delete either single or multiple records another screen will appear for confirmation of deletion.

Are you sure?

Are you sure you want to delete the selected Student's Information? All of the following objects and their related items will be deleted:

**Summary**

- Student's Information: 2

**Objects**

- Student Information: [Mr. India](#)
- Student Information: [Missile Man](#)

## django.contrib package

Django aims to follow Python's "batteries included" philosophy. It ships with a variety of extra, optional tools that solve common Web-development problems.

This code lives in **django.contrib** in the Django distribution. This document gives a rundown of the packages in **contrib**, along with any dependencies those packages have.

- The Django admin site
- **django.contrib.auth**
- The contenttypes framework
- The flatpages app
- GeoDjango
- **django.contrib.humanize**
- The messages framework
- **django.contrib.postgres**
- The redirects app
- The sitemap framework
- The "sites" framework
- The **staticfiles** app
- The syndication feed framework

**admin:** The automatic Django administrative interface. Requires the auth and contenttypes contrib packages to be installed.

**auth:** Django's authentication framework.

**contenttypes:** A light framework for hooking into "types" of content, where each installed Django model is a separate content type.

**flatpages:** A framework for managing simple "flat" HTML content in a database. Requires the sites contrib package to be installed as well.

**gis:** A world-class geospatial framework built on top of Django, that enables storage, manipulation and display of spatial data.

**humanize:** A set of Django template filters useful for adding a "human touch" to data.

**messages:** A framework for storing and retrieving temporary cookie- or session-based messages

**postgres:** A collection of PostgreSQL specific features.

**redirects:** A framework for managing redirects.

**sessions:** A framework for storing data in anonymous sessions.

**sites:** A light framework that lets you operate multiple websites off of the same database and Django installation. It gives you hooks for associating objects to one or more sites.

**sitemaps:** A framework for generating Google sitemap XML files.

**syndication:** A framework for generating syndication feeds, in RSS and Atom, quite easily.

## Form Basics

### HTML forms

In HTML, a form is a collection of elements inside `<form>...</form>` that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

Some of these form interface elements - text input or checkboxes - are fairly simple and are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form `<input>` elements to achieve these effects.

As well as its `<input>` elements, a form must specify two things:

- *where*: the URL to which the data corresponding to the user's input should be returned
- *how*: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several `<input>` elements: one of `type="text"` for the username, one of `type="password"` for the password, and one of `type="submit"` for the "Log in" button. It also contains some hidden text fields that the user doesn't see, which Django uses to determine what to do next.

It also tells the browser that the form data should be sent to the URL specified in the `<form>`'s `action` attribute - `/admin/` - and that it should be sent using the HTTP mechanism specified by the `method` attribute - `post`.

When the `<input type="submit" value="Log in">` element is triggered, the data is returned to `/admin/`.

## GET and POST

**GET** and **POST** are the only HTTP methods to use when dealing with forms. Django's login form is returned using the **POST** method, in which the browser bundles up the form data, encodes it for transmission, sends it to the server, and then receives back its response.

**GET**, by contrast, 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. You can see this in action if you do a search in the Django documentation, which will produce a URL of the form <https://docs.djangoproject.com/search/?q=forms&release=1>.

**GET** and **POST** are typically used for different purposes.

Any request that could be used to change the state of the system - for example, a request that makes changes in the database - should use **POST**. **GET** should be used only for requests that do not affect the state of the system.

**GET** would also be unsuitable for a password form, because the password would appear in the URL, and thus, also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of data, or for binary data, such as an image. A Web application that uses **GET** requests for admin forms is a security risk: it can be easy for an attacker to mimic a form's request to gain access to sensitive parts of the system. **POST**, coupled with other protections like **Django's CSRF protection** offers more control over access.

On the other hand, **GET** is suitable for things like a web search form, because the URLs that represent a **GET** request can easily be bookmarked, shared, or resubmitted.



## Form Validation

Form validation happens when the data is cleaned. If you want to customize this process, there are various places to make changes, each one serving a different purpose. Three types of cleaning methods are run during form processing. These are normally executed when you call the **is\_valid()** method on a form. There are other things that can also trigger cleaning and validation (accessing the **errors** attribute or calling **full\_clean()** directly), but normally they won't be needed.

In general, any cleaning method can raise **ValidationError** if there is a problem with the data it is processing, passing the relevant information to the **ValidationError** constructor. If no **ValidationError** is raised, the method should return the cleaned (normalized) data as a Python object.

Most validation can be done using validators - simple helpers that can be reused easily. Validators are simple functions (or callables) that take a single argument and raise **ValidationError** on invalid input. Validators are run after the field's **to\_python** and **validate** methods have been called.

Validation of a form is split into several steps, which can be customized or overridden:

- The **to\_python()** method on a **Field** is the first step in every validation. It coerces the value to a correct datatype and raises **ValidationError** if that is not possible. This method accepts the raw value from the widget and returns the converted value. For example, a **FloatField** will turn the data into a Python **float** or raise a **ValidationError**.
- The **validate()** method on a **Field** handles field-specific validation that is not suitable for a validator. It takes a value that has been coerced to a correct datatype and raises **ValidationError** on any error. This method does not return anything and shouldn't alter the value. You should override it to handle validation logic that you can't or don't want to put in a validator.
- The **run\_validators()** method on a **Field** runs all of the field's validators and aggregates all the errors into a single **ValidationError**. You shouldn't need to override this method.
- The **clean()** method on a **Field** subclass is responsible for running **to\_python()**, **validate()**, and **run\_validators()** in the correct order and propagating their errors. If, at any time, any of the methods raise **ValidationError**, the validation stops and that error is raised. This method returns the clean data, which is then inserted into the **cleaned\_data** dictionary of the form.

## Rendering forms

### The Form class

We already know what we want our HTML form to look like. Our starting point for it in Django is this:

#### #forms.py

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

This defines a **Form** class with a single field (**your\_name**). We've applied a human-friendly label to the field, which will appear in the **<label>** when it's rendered (although in this case, the **label** we specified is actually the same one that would be generated automatically if we had omitted it).

The field's maximum allowable length is defined by **max\_length**. This does two things. It puts a **maxlength="100"** on the HTML **<input>** (so the browser should prevent the user from entering more than that number of characters in the first place). It also means that when Django receives the form back from the browser, it will validate the length of the data.

A **Form** instance has an **is\_valid()** method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:

- return **True**
- place the form's data in its **cleaned\_data** attribute.

The whole form, when rendered for the first time, will look like:

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

Note that it **does not** include the **<form>** tags, or a submit button. We'll have to provide those ourselves in the template.

### The view

Form data sent back to a Django website is processed by a view, generally the same view which published the form. This allows us to reuse some of the same logic. To handle the form we need to instantiate it in the view for the URL where we want it to be published:

#### #views.py

```
from django.http import HttpResponseRedirect

from django.shortcuts import render
from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
```

```

if request.method == 'POST':
    # create a form instance and populate it with data from the request:
    form = NameForm(request.POST)
    # check whether it's valid:
    if form.is_valid():
        # process the data in form.cleaned_data as required
        # redirect to a new URL:
        return HttpResponseRedirect('/thanks/')
    # if a GET (or any other method) we'll create a blank form
else:
    form = NameForm()
return render(request, 'name.html', {'form': form})

```

If we arrive at this view with a **GET** request, it will create an empty form instance and place it in the template context to be rendered. This is what we can expect to happen the first time we visit the URL.

If the form is submitted using a **POST** request, the view will once again create a form instance and populate it with data from the request: **form = NameForm(request.POST)** This is called “binding data to the form” (it is now a *bound* form).

We call the form’s **is\_valid()** method; if it’s not **True**, we go back to the template with the form. This time the form is no longer empty (*unbound*) so the HTML form will be populated with the data previously submitted, where it can be edited and corrected as required.

If **is\_valid()** is **True**, we’ll now be able to find all the validated form data in its **cleaned\_data** attribute. We can use this data to update the database or do other processing before sending an HTTP redirect to the browser telling it where to go next.

### The template

We don’t need to do much in our **name.html** template. The simplest example is:

```

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

```

All the form’s fields and their attributes will be unpacked into HTML markup from that **{{ form }}** by Django’s template language.

### Widgets

Each form field has a corresponding Widget class, which in turn corresponds to an HTML form widget such as **<input type="text">**.

In most cases, the field will have a sensible default widget. For example, by default, a **CharField** will have a **TextInput** widget, that produces an **<input type="text">** in the HTML. If you needed **<textarea>** instead, you’d specify the appropriate widget when defining your form field, as we have done for the **message** field.

### Working with form templates

All you need to do to get your form into a template is to place the form instance into the template context. So if your form is called form in the context, `{{ form }}` will render its `<label>` and `<input>` elements appropriately.

There are other output options though for the `<label>/<input>` pairs:

```

{{ form.as_table }} will render them as table cells wrapped in <tr> tags
{{ form.as_p }} will render them wrapped in <p> tags
{{ form.as_ul }} will render them wrapped in <li> tags

```

Note that you'll have to provide the surrounding `<table>` or `<ul>` elements yourself. Here's the output of `{{ form.as_p }}` for our ContactForm instance:

```

<p><label for="id_subject">Subject:</label>
  <input id="id_subject" type="text" name="subject" maxlength="100" required></p>
<p><label for="id_message">Message:</label>
  <textarea name="message" id="id_message" required></textarea></p>
<p><label for="id_sender">Sender:</label>
  <input type="email" name="sender" id="id_sender" required></p>
<p><label for="id_cc_myself">Cc myself:</label>
  <input type="checkbox" name="cc_myself" id="id_cc_myself"></p>

```

### Rendering fields manually

We don't have to let Django unpack the form's fields; we can do it manually if we like (allowing us to reorder the fields, for example). Each field is available as an attribute of the form using `{{ form.name_of_field }}`, and in a Django template, will be rendered appropriately. For example:

```

{{ form.non_field_errors }}
<div class="fieldWrapper">
  {{ form.subject.errors }}
  <label for="{{ form.subject.id_for_label }}">Email subject:</label>
  {{ form.subject }}
</div>
<div class="fieldWrapper">
  {{ form.message.errors }}
  <label for="{{ form.message.id_for_label }}">Your message:</label>
  {{ form.message }}
</div>
<div class="fieldWrapper">
  {{ form.sender.errors }}
  <label for="{{ form.sender.id_for_label }}">Your email address:</label>
  {{ form.sender }}
</div>
<div class="fieldWrapper">
  {{ form.cc_myself.errors }}
  <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
  {{ form.cc_myself }}
</div>

```

## ModelForm

### **class ModelForm**

If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a **BlogComment** model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model.

For this reason, Django provides a helper class that lets you create a **Form** class from a Django model.

### Field types

The generated **Form** class will have a form field for every model field specified, in the order specified in the **fields** attribute.

Each model field has a corresponding default form field. For example, a **CharField** on a model is represented as a **CharField** on a form. A model **ManyToManyField** is represented as a **MultipleChoiceField**. Here is the full list of conversions:

Model field	Form field
<b>AutoField</b>	Not represented in the form
<b>BigAutoField</b>	Not represented in the form
<b>BigIntegerField</b>	IntegerField with <code>min_value</code> set to <code>-9223372036854775808</code> and <code>max_value</code> set to <code>9223372036854775807</code> .
<b>BinaryField</b>	CharField, if <code>editable</code> is set to <code>True</code> on the model field, otherwise not represented in the form.
<b>BooleanField</b>	BooleanField, or <code>NullBooleanField</code> if <code>null=True</code> .
<b>CharField</b>	CharField with <code>max_length</code> set to the model field's <code>max_length</code> and <code>empty_value</code> set to <code>None</code> if <code>null=True</code> .
<b>.DateField</b>	DateField
<b>DateTimeField</b>	DateTimeField
<b>DecimalField</b>	DecimalField
<b>EmailField</b>	EmailField
<b>FileField</b>	FileField
<b>FilePathField</b>	FilePathField
<b>FloatField</b>	FloatField
<b>ForeignKey</b>	ModelChoiceField
<b>ImageField</b>	ImageField
<b>IntegerField</b>	IntegerField
<b>IPAddressField</b>	IPAddressField
<b>GenericIPAddressField</b>	GenericIPAddressField
<b>ManyToManyField</b>	ModelMultipleChoiceField
<b>NullBooleanField</b>	NullBooleanField
<b>PositiveIntegerField</b>	IntegerField
<b>PositiveSmallIntegerField</b>	IntegerField
<b>SlugField</b>	SlugField
<b>SmallIntegerField</b>	IntegerField
<b>TextField</b>	CharField with <code>widget=forms.Textarea</code>
<b>TimeField</b>	TimeField
<b>URLField</b>	URLField

As you might expect, the **ForeignKey** and **ManyToManyField** model field types are special cases:

- **ForeignKey** is represented by `django.forms.ModelChoiceField`, which is a **ChoiceField** whose choices are a model **QuerySet**.
- **ManyToManyField** is represented by `django.forms.ModelMultipleChoiceField`, which is a **MultipleChoiceField** whose choices are a model **QuerySet**.

In addition, each generated form field has attributes set as follows:

- If the model field has **blank=True**, then **required** is set to **False** on the form field. Otherwise, **required=True**.
- The form field's **label** is set to the **verbose\_name** of the model field, with the first character capitalized.
- The form field's **help\_text** is set to the **help\_text** of the model field.
- 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**. The choices will normally include the blank choice which is selected by default. If the field is required, this forces the user to make a selection. The blank choice will not be included if the model field has **blank=False** and an explicit **default** value (the **default** value will be initially selected instead).

### A full example

Consider this set of models:

```
class dataForm(forms.ModelForm):
    rname = forms.CharField(max_length=150,
        label="Student Name",
        min_length=10,
        widget=forms.Textarea
    )
    class Meta:
        model = studInfo
        fields = '__all__'
```

### Selecting the fields to use

It is strongly recommended that you explicitly set all fields that should be edited in the form using the **fields** attribute. Failure to do so can easily lead to security problems when a form unexpectedly allows a user to set certain fields, especially when new fields are added to a model. Depending on how the form is rendered, the problem may not even be visible on the web page.

The alternative approach would be to include all fields automatically, or blacklist only some. This fundamental approach is known to be much less secure and has led to serious exploits on major websites (e.g. GitHub).



There are, however, two shortcuts available for cases where you can guarantee these security concerns do not apply to you:

Set the **fields** attribute to the special value '**\_\_all\_\_**' to indicate that all fields in the model should be used. For example:

```
from django.forms import ModelForm
class AuthorForm(ModelForm):
    class Meta:
        model = studInfo
        fields = '__all__'
```

Set the **exclude** attribute of the **ModelForm**'s inner **Meta** class to a list of fields to be excluded from the form.

**Example:**

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = studInfo
        exclude = ['title']
```

Since the **studInfo** model has the 3 fields **name**, **title** and **birth\_date**, this will result in the fields **name** and **birth\_date** being present on the form.

If either of these are used, the order the fields appear in the form will be the order the fields are defined in the model, with **ManyToManyField** instances appearing last.

In addition, Django applies the following rule: if you set **editable=False** on the model field, *any* form created from the model via **ModelForm** will not include that field.

## The View Layer

Django has the concept of “**views**” to encapsulate the logic responsible for processing a user’s request and for returning the response.

- **The basics:** URLconfs | View functions | Shortcuts | Decorators
- **Reference:** Built-in Views | Request/response objects | TemplateResponse objects
- **File uploads:** Overview | File objects | Storage API | Managing files | Custom storage
- **Class-based views:** Overview | Built-in display views | Built-in editing views | Using mixins | API reference | Flattened index
- **Advanced:** Generating CSV | Generating PDF
- **Middleware:** Overview | Built-in middleware classes

### URL dispatcher

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations.

See Cool URIs don’t change, by World Wide Web creator Tim Berners-Lee, for excellent arguments on why URLs should be clean and usable.

### Writing views

A view function, or *view* for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it’s on your Python path. There’s no other requirement—no “**magic**,” so to speak. For the sake of putting the code *somewhere*, the convention is to put views in a file called **views.py**, placed in your project or application directory.

### Django shortcut functions

The package **django.shortcuts** collects helper functions and classes that “**span**” multiple levels of MVC. In other words, these functions/classes introduce controlled coupling for convenience’s sake.

- `render()`
- `render_to_response`
- `redirect`
- `get_object_or_404`
- `get_list_or_404`

### Built-In Views

- ❖ Error views
  - The **404** (page not found) view
  - The **500** (server error) view
  - The **403** (HTTP Forbidden) view
  - The **400** (bad request) view

### Request and Response objects

Django uses request and response objects to pass state through the system. When a page is requested, Django creates an **HttpRequest** object that contains metadata about the request. Then Django loads the appropriate view, passing the **HttpRequest** as the first argument to the view function. Each view is responsible for returning an **HttpResponse** object.

### File Uploads

When Django handles a file upload, the file data ends up placed in **request.FILES** (for more on the **request** object see the documentation for request and response objects). This document explains how files are stored on disk and in memory, and how to customize the default behavior.

### Class-based views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for simple tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case

At its core, a class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

### Outputting PDFs with Django

This is made possible by the excellent, open-source **ReportLab** Python PDF library. The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes – say, for different users or different pieces of content.

### Middleware

Middleware is a framework of hooks into Django's request/response processing. It's a light, low-level "plugin" system for globally altering Django's input or output.

Each middleware component is responsible for doing some specific function. For example, Django includes a middleware component, **AuthenticationMiddleware**, that associates users with requests using sessions.

This document explains how middleware works, how you activate middleware, and how to write your own middleware. Django ships with some built-in middleware you can use right out of the box.

## Requesting a Web Page via URL

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations.

To design URLs for an app, you create a Python module informally called a **URLconf** (URL configuration). This module is pure Python code and is a simple mapping between URL patterns (simple regular expressions) to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it's pure Python code, it can be constructed dynamically.

### How Django processes a request

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

1. Django determines the root URLconf module to use. Ordinarily, this is the value of the **ROOT\_URLCONF** setting, but if the incoming **HttpRequest** object has an attribute called **urlconf** (set by middleware request processing), its value will be used in place of the **ROOT\_URLCONF** setting.
2. Django loads that Python module and looks for the variable **urlpatterns**. This should be a Python list, in the format returned by the function **django.conf.urls.patterns()**.
3. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL.
4. Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function (or a class based view). The view gets passed the following arguments:
  - An instance of **HttpRequest**.
  - If the matched regular expression returned no named groups, then the matches from the regular expression are provided as positional arguments.
  - The keyword arguments are made up of any named groups matched by the regular expression, overridden by any arguments specified in the optional **kwargs** argument to **django.conf.urls.url()**.
5. If no regex matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view. See Error handling below.

### Example

```
from django.conf.urls import patterns, url
from . import views
urlpatterns = patterns("",
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/(\d{4})/$', views.year_archive),
    url(r'^articles/(\d{4})/(\d{2})/$', views.month_archive),
    url(r'^articles/(\d{4})/(\d{2})/(\d+)/$', views.article_detail),
)
```

## Notes:

- To capture a value from the URL, just put parenthesis around it.
- There's no need to add a leading slash, because every URL has that. For example, it's **^articles**, not **^/articles**.
- The 'r' in front of each regular expression string is optional but recommended. It tells Python that a string is "raw" – that nothing in the string should be escaped.

A convenient trick is to specify default parameters for your views' arguments. Here's an example URLconf and view:

```
# URLconf
from django.conf.urls import patterns, url
from . import views
urlpatterns = patterns('',
    url(r'^blog/$', views.page),
    url(r'^blog/page(?P<num>\d+)/$', views.page),
)
```

```
# View (in blog/views.py)
def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    ...
```

In the above example, both URL patterns point to the same view – `views.page` – but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, "1". If the second pattern matches, `page()` will use whatever `num` value was captured by the regex.

**The view prefix**

If you do use strings, it is possible to specify a common prefix in your `patterns()` call.

```
from django.conf.urls import patterns, url
urlpatterns = patterns('',
    url(r'^articles/(\d{4})/$', 'news.views.year_archive'),
    url(r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
    url(r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

**Multiple view prefixes**

In practice, you'll probably end up mixing and matching views to the point where the views in your `urlpatterns` won't have a common prefix. However, you can still take advantage of the view prefix shortcut to remove duplication.

```
from django.conf.urls import patterns, url
urlpatterns = patterns('myapp.views',
    url(r'^$', 'app_index'),
    url(r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$', 'month_display'),
)
urlpatterns += patterns('weblog.views',
    url(r'^tag/(?P<tag>\w+)/$', 'tag'),
)
```

## Rendering Web Page via View Function

A view function, or *view* for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path. There's no other requirement—no “magic,” so to speak. For the sake of putting the code *somewhere*, the convention is to put views in a file called **views.py**, placed in your project or application directory.

### A simple view

Here's a view that returns the current date and time, as an HTML document:

```
from django.http import HttpResponse
import datetime

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

Let's step through this code one line at a time:

- First, we import the class **HttpResponse** from the **django.http** module, along with Python's **datetime** library.
- Next, we define a function called **current\_datetime**. This is the view function. Each view function takes an **HttpRequest** object as its first parameter, which is typically named **request**.

Note that the name of the view function doesn't matter; it doesn't have to be named in a certain way in order for Django to recognize it. We're calling it **current\_datetime** here, because that name clearly indicates what it does.

- The view returns an **HttpResponse** object that contains the generated response. Each view function is responsible for returning an **HttpResponse** object. (There are exceptions, but we'll get to those later.)

### Returning errors

Returning HTTP error codes in Django is easy. There are subclasses of **HttpResponse** for a number of common HTTP status codes other than 200 (which means “OK”). You can find the full list of available subclasses in the request/response documentation. Just return an instance of one of those subclasses instead of a normal **HttpResponse** in order to signify an error. For example:

```
from django.http import HttpResponse, HttpResponseNotFound
def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```



There isn't a specialized subclass for every possible HTTP response code, since many of them aren't going to be that common. However, as documented in the **HttpResponse** documentation, you can also pass the HTTP status code into the constructor for **HttpResponse** to create a return class for any status code you like. For example:

```
from django.http import HttpResponse
```

```
def my_view(request):
    # ...
    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

Because 404 errors are by far the most common HTTP error, there's an easier way to handle those errors.

### The **Http404** exception

```
class django.http.Http404
```

When you return an error such as **HttpResponseNotFound**, you're responsible for defining the HTML of the resulting error page:

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

For convenience, and because it's a good idea to have a consistent 404 error page across your site, Django provides an **Http404** exception. If you raise **Http404** at any point in a view function, Django will catch it and return the standard error page for your application, along with an HTTP error code 404.

#### Example:

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

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

In order to show customized HTML when Django returns a 404, you can create an HTML template named **404.html** and place it in the top level of your template tree. This template will then be served when **DEBUG** is set to **False**.

When **DEBUG** is **True**, you can provide a message to **Http404** and it will appear in the standard 404 debug template. Use these messages for debugging purposes; they generally aren't suitable for use in a production 404 template.

## Render HTTPResponse to Templates

Standard **HttpResponse** objects are static structures. They are provided with a block of pre-rendered content at time of construction, and while that content can be modified, it isn't in a form that makes it easy to perform modifications.

However, it can sometimes be beneficial to allow decorators or middleware to modify a response *after* it has been constructed by the view. For example, you may want to change the template that is used, or put additional data into the context.

**TemplateResponse** provides a way to do just that. Unlike basic **HttpResponse** objects, **TemplateResponse** objects retain the details of the template and context that was provided by the view to compute the response. The final output of the response is not computed until it is needed, later in the response process.

**TemplateResponse** objects

### *class* **TemplateResponse**

**TemplateResponse** is a subclass of **SimpleTemplateResponse** that uses a **RequestContext** instead of a **Context**.

#### Methods

```
TemplateResponse.__init__(request,
template,
context=None,
content_type=None,
status=None,
current_app=None)
```

Instantiates an **TemplateResponse** object with the given template, context, MIME type and HTTP status.

**request** : An **HttpRequest** instance.

**template**:The full name of a template, or a sequence of template names. **Template** instances can also be used.

**context**: A dictionary of values to add to the template context. By default, this is an empty dictionary. **Context** objects are also accepted as **context** values. If you pass a **Context** instance or subclass, it will be used instead of creating a new **RequestContext**.

**status**: The HTTP Status code for the response.

**content\_type** : The value included in the HTTP **Content-Type** header, including the MIME type specification and the character set encoding. If **content\_type** is specified, then its value is used. Otherwise, **DEFAULT\_CONTENT\_TYPE** is used.

**current\_app**: A hint indicating which application contains the current view. See the namespaced URL resolution strategy for more information.

## Understanding Context Data and Python Dictionary Type

When you use a Django Template, it is compiled once (and only once) and stored for future use, as an optimization. A template can have variable names in double curly braces, such as `{{ myvar1 }}` and `{{ myvar2 }}`.

A **Context** is a dictionary with variable names as the **key** and their values as the **value**. Hence, if your context for the above template looks like: `{myvar1: 101, myvar2: 102}`, when you pass this context to the template render method, `{{ myvar1 }}` would be replaced with 101 and `{{ myvar2 }}` with 102 in your template. This is a simplistic example, but really a Context object is the *context* in which the template is being rendered.

As for a ContextProcessor, that is a slightly advanced concept. You can have in your settings.py file listed a few Context Processors which take in an HttpRequest object and return a dictionary (similar to the Context object above). The dictionary (context) returned by the Context Processor is merged into the context passed in by you (the user) by Django.

A use case for a Context Processor is when you always want to insert certain variables inside your template (for example the location of the user could be a candidate). Instead of writing code to insert it in each view, you could simply write a context processor for it and add it to the `TEMPLATE_CONTEXT_PROCESSORS` settings in settings.py.

### Rendering a context

Once you have a compiled **Template** object, you can render a context with it. You can reuse the same template to render it several times with different contexts.

### *class* Context(dict\_ =None)

The constructor of `django.template.Context` takes an optional argument — a dictionary mapping variable names to variable values.

### Template.render(context)

Call the **Template** object's `render()` method with a **Context** to “fill” the template:

```
>>> from django.template import Context, Template
>>> template = Template("My name is {{ my_name }}.")

>>> context = Context({"my_name": "Adrian"})
>>> template.render(context)
"My name is Adrian."

>>> context = Context({"my_name": "Dolores"})
>>> template.render(context)
"My name is Dolores."
```

## Cookies: Getting and Setting Cookies

A cookie is a small piece of information which is stored in the client browser. It is used to store user's data in a file permanently (or for the specified time). Cookie has its expiry date and time and removes automatically when gets expire. Django provides built-in methods to set and fetch cookie.

The **set\_cookie()** method is used to set a cookie and **get()** method is used to get the cookie. The **request.COOKIES['key']** array can also be used to get cookie values.

Django Cookie Example

In **views.py**, two functions **setcookie()** and **getcookie()** are used to set and get cookie respectively

```
// views.py
from django.shortcuts import render
from django.http import HttpResponseRedirect

def setcookie(request):
    response = HttpResponseRedirect("Cookie Set")
    response.set_cookie('college', 'Geetanjali')
    return response
def getcookie(request):
    clname = request.COOKIES['college']
    return HttpResponseRedirect("College: "+ clname );
```

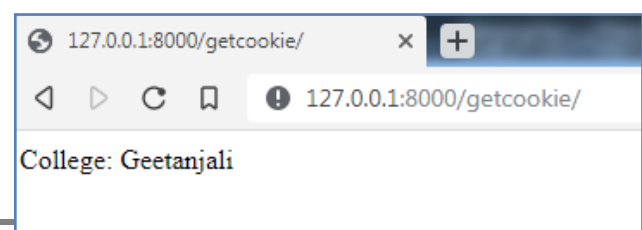
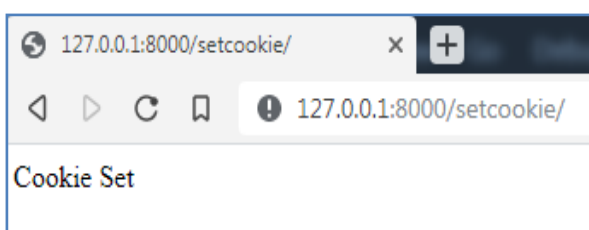
And URLs specified to access these functions.

```
// urls.py
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path(r'^admin/', admin.site.urls),
    path(r'^index/', views.index),
    path(r'^setcookie',views.setcookie,name='setcookie'),
    path(r'^getcookie',views.getcookie,name='getcookie')
]
```

### Start Server

```
$ python3 manage.py runserver
```

After starting the server, set cookie by using **localhost:8000/setcookie** URL. It shows the following output to the browser.



## Session: Django's session

A session is a mechanism to store information on the server side during the interaction with the web application.

In Django, by default session stores in the database and also allows file-based and cache based sessions. It is implemented via a piece of middleware and can be enabled by using the following code.

Put `django.contrib.sessions.middleware.SessionMiddleware` in `MIDDLEWARE` and `django.contrib.sessions` in `INSTALLED_APPS` of settings.py file.

To set and get the session in views, we can use `request.session` and can set multiple times too.

The class `backends.base.SessionBase` is a base class of all session objects. It contains the following standard methods.

Method	Description
<code>__getitem__(key)</code>	It is used to get session value.
<code>__setitem__(key, value)</code>	It is used to set session value.
<code>__delitem__(key)</code>	It is used to delete session object.
<code>__contains__(key)</code>	It checks whether the container contains the particular session object or not.
<code>get(key, default=None)</code>	It is used to get session value of the specified key.

Let's see an example in which we will set and get session values. Two functions are defined in the `views.py` file.

The first function is used to set and the second is used to get session values.

```
//views.py
from django.shortcuts import render
from django.http import HttpResponse

def setsession(request):
    request.session['lecturer'] = 'HardikChavda'
    request.session['semail'] = 'hardikkchavda@gmail.com'
    return HttpResponse("Session is Set")
def getsession(request):
    sname = request.session['lecturer']
    semail = request.session['semail']
    return HttpResponse(sname + " " + semail);
```

Url mapping to call both the functions.

```
// urls.py
from django.contrib import admin
from django.urls import path
```

```

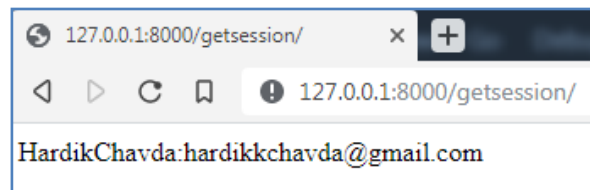
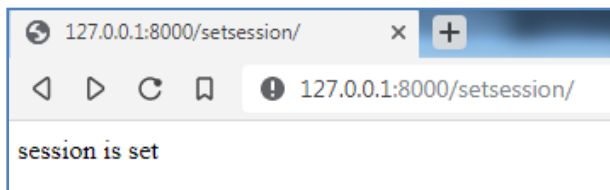
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
    path('session',views.setsession),
    path('gsession',views.getsession)
]

```

### Run Server

```
> python manage.py runserver
```

And set the session by using **localhost:8000/setsession**



## Session Outside Views

```

>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore()
>>> # stored as seconds since epoch since datetimes are not serializable in JSON.
>>> s['last_login'] = 1376587691
>>> s.create()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceed'
>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceed')
>>> s['last_login']
1376587691

```

**SessionStore.create()** is designed to create a new session (i.e. one not loaded from the session store and with **session\_key=None**). **save()** is designed to save an existing session (i.e. one loaded from the session store). Calling **save()** on a new session may also work but has a small chance of generating a **session\_key** that collides with an existing one. **create()** calls **save()** and loops until an unused **session\_key** is generated.



## Testing Django

Testing is an important but often neglected part of any Django project. In this tutorial we'll review testing best practices and example code that can be applied to any Django app.

Broadly speaking there are two types of tests you need to run:

- **Unit Tests** are small, isolated, and focus on one specific function.
- **Integration Tests** are aimed at mimicking user behavior and combine multiple pieces of code and functionality.

While we might use a *unit test* to confirm that the homepage returns an HTTP status code of 200, an *integration test* might mimic the entire registration flow of a user. For all tests the expectation is that the result is either expected, unexpected, or an error. An expected result would be a 200 response on the homepage, but we can—and should—also test that the homepage does not return something unexpected, like a 404 response. Anything else would be an error requiring further debugging.

The main focus of testing should be unit tests. You can't write too many of them. They are far easier to write, read, and debug than integration tests. They are also quite fast to run.

### When to run tests

The short answer is **all the time!** Practically speaking whenever code is pushed or pulled from a repo to a staging environment is ideal. A continuous integration service can perform this automatically. You should also re-run all tests when upgrading software packages, especially Django itself.

### Layout

By default all new apps in a Django project come with a tests.py file. Any test within this file that starts with test\_ will be run by Django's test runner. **Make sure all test files start with test\_.**

As projects grow in complexity, it's recommended to delete this initial tests.py file and replace it with an app-level tests folder that contains individual tests files for each area of functionality.

Example:

```
|__app
|__tests
|-- __init__.py
|-- test_forms.py
|-- test_models.py
|-- test_views.py
```

On the command line run the following commands to start our new project. We'll place the code in a folder called testy on the Desktop, but you can locate the code anywhere you choose.

```
(testy) D:\Python\testy> django-admin startproject myproject .
(testy) D:\Python\testy> python manage.py startapp pages
```

Now update settings.py to add our new pages app and configure Django to look for a project-level templates folder.

```
# myproject/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    .
    .
    'pages.apps.PagesConfig', # new
]
```

```
TEMPLATES = [
    ...
    'DIRS': [os.path.join(BASE_DIR, 'templates')], # new
    ...
]
```

Create our two templates to test for a homepage and about page.

```
(testy) D:\Python\testy> mkdir templates
(testy) D:\Python\testy> cd templates/home.html
(testy) D:\Python\testy> cd templates/about.html
```

Populate the templates with the following simple code.

```
<!-- templates/home.html -->
<h1>Homepage</h1>
<!-- templates/about.html -->
<h1>About page</h1>
```

Update the project-level urls.py file to point to the pages app.

```
# myproject/urls.py
from django.contrib import admin
from django.urls import path, include # new
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')), # new
]
```

Create a urls.py file within the pages app.

```
(testy) D:\Python\testy> cd pages/urls.py
```

Then update it as follows:

```
# pages/urls.py  
from django.urls import path  
from .views import HomePageView, AboutPageView  
urlpatterns = [  
    path('', HomePageView.as_view(), name='home'),  
    path('about/', AboutPageView.as_view(), name='about'),  
]
```

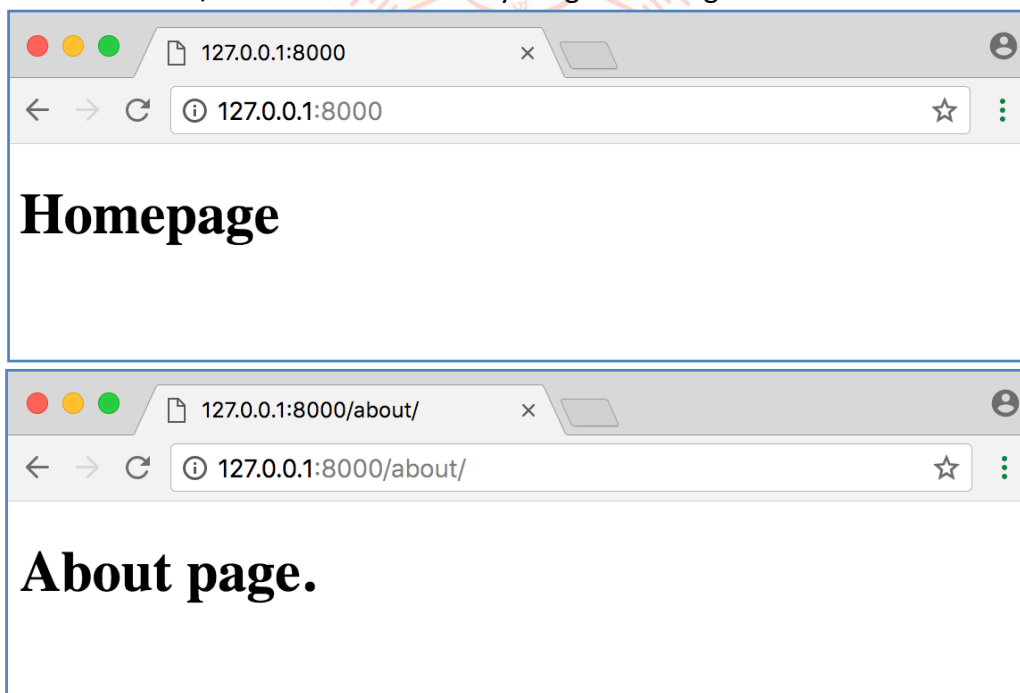
And as a final step add our views.

```
# pages/views.py  
from django.views.generic import TemplateView  
class HomePageView(TemplateView):  
    template_name = 'home.html'  
class AboutPageView(TemplateView):  
    template_name = 'about.html'
```

Start up the local Django server.

```
(testy) D:\Python\testy> python manage.py runserver
```

Then navigate to the homepage at <http://127.0.0.1:8000/> and about page at <http://127.0.0.1:8000/about/> to confirm everything is working.



**Time for tests.****SimpleTestCase**

Our Django application only has two static pages at the moment. There's no database involved which means we should use SimpleTestCase.

We can use the existing pages/tests.py file for our tests for now. Take a look at the code below which adds five tests for our homepage. First we test that it exists and returns a 200 HTTP status code. Then we confirm that it uses the url named home. We check that the template used is home.html, the HTML matches what we've typed so far, and even test that it does not contain incorrect HTML. It's always good to test both **expected** and **unexpected** behavior.

```
# pages/tests.py
from django.http import HttpRequest
from django.test import SimpleTestCase
from django.urls import reverse

from pages import views
class HomePageTests(SimpleTestCase):

    def test_home_page_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_view_url_by_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_view_uses_correct_template(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')

    def test_home_page_contains_correct_html(self):
        response = self.client.get('/')
        self.assertContains(response, '<h1>Homepage</h1>')

    def test_home_page_does_not_contain_incorrect_html(self):
        response = self.client.get('/')
        self.assertNotContains(
            response, 'Hi there! I should not be on the page.')
```

Now run the tests.

```
(testy) D:\Python\testy> python manage.py test
```

They should all pass.

As an exercise, see if you can add a class for `AboutPageTests` in this same file. It should have the same five tests but will need to be updated slightly. Run the test runner once complete. The correct code is below so try not to peak...

```
# pages/tests.py  
class AboutPageTests(SimpleTestCase):  
  
    def test_about_page_status_code(self):  
        response = self.client.get('/about/')  
        self.assertEqual(response.status_code, 200)  
  
    def test_view_url_by_name(self):  
        response = self.client.get(reverse('about'))  
        self.assertEqual(response.status_code, 200)  
  
    def test_view_uses_correct_template(self):  
        response = self.client.get(reverse('about'))  
        self.assertEqual(response.status_code, 200)  
        self.assertTemplateUsed(response, 'about.html')  
  
    def test_about_page_contains_correct_html(self):  
        response = self.client.get('/about/')  
        self.assertContains(response, '<h1>About page</h1>')  
  
    def test_about_page_does_not_contain_incorrect_html(self):  
        response = self.client.get('/')  
        self.assertNotContains(  
            response, 'Hi there! I should not be on the page.')
```

### Message Board app

Now let's create our *message board* app so we can try testing out database queries. First create another app called posts.

```
(testy) D:\Python\testy> python manage.py startapp posts
```

Add it to our settings.py file.

```
# myproject/settings.py  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',
```

```
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'pages.apps.PagesConfig',
'posts.apps.PostsConfig', # new
]
```

Then run migrate to create our initial database.

```
(testy) D:\Python\testy> python manage.py migrate
```

Now add a basic model.

```
# posts/models.py
from django.db import models
class Post(models.Model):
    text = models.TextField()
    def __str__(self):
        """A string representation of the model."""
        return self.text
```

Create a database migration file and activate it.

```
(testy) D:\Python\testy> python manage.py makemigrations posts
(testy) D:\Python\testy> python manage.py migrate posts
```

For simplicity we can just a post via the Django admin. So first create a superuser account and fill in all prompts.

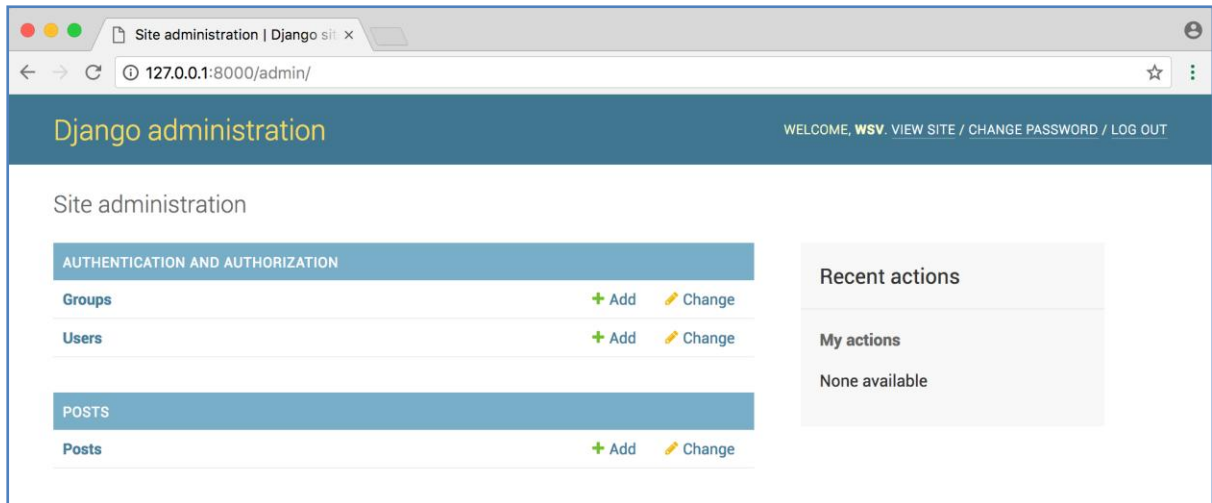
```
(testy) D:\Python\testy> python manage.py createsuperuser
```

Update our admin.py file so the posts app is active in the Django admin.

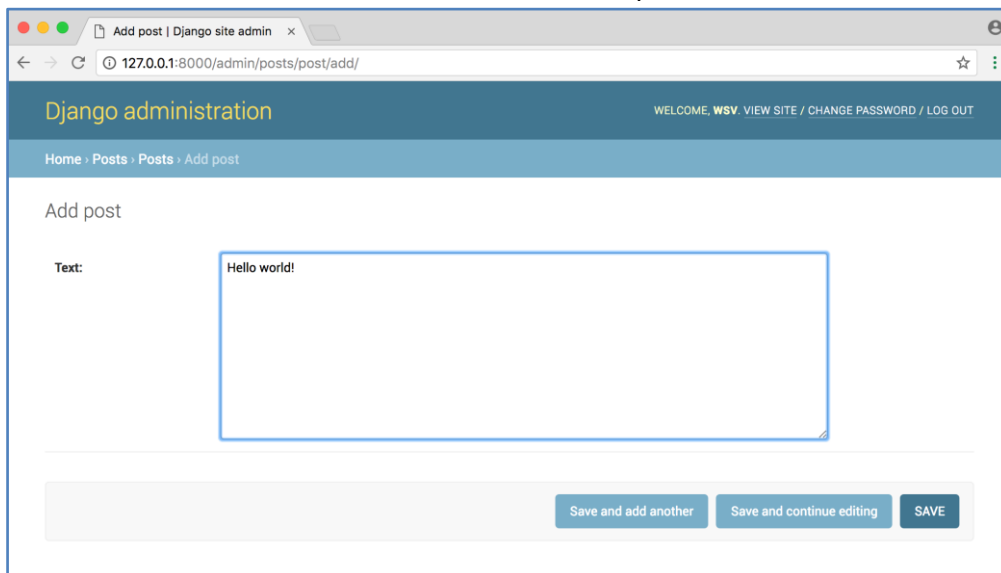
```
# posts/admin.py
from django.contrib import admin
from .models import Post
admin.site.register(Post)
```

Then restart the Django server with `python manage.py runserver` and login to the Django admin at <http://127.0.0.1:8000/admin/>. You should see the admin's login screen:

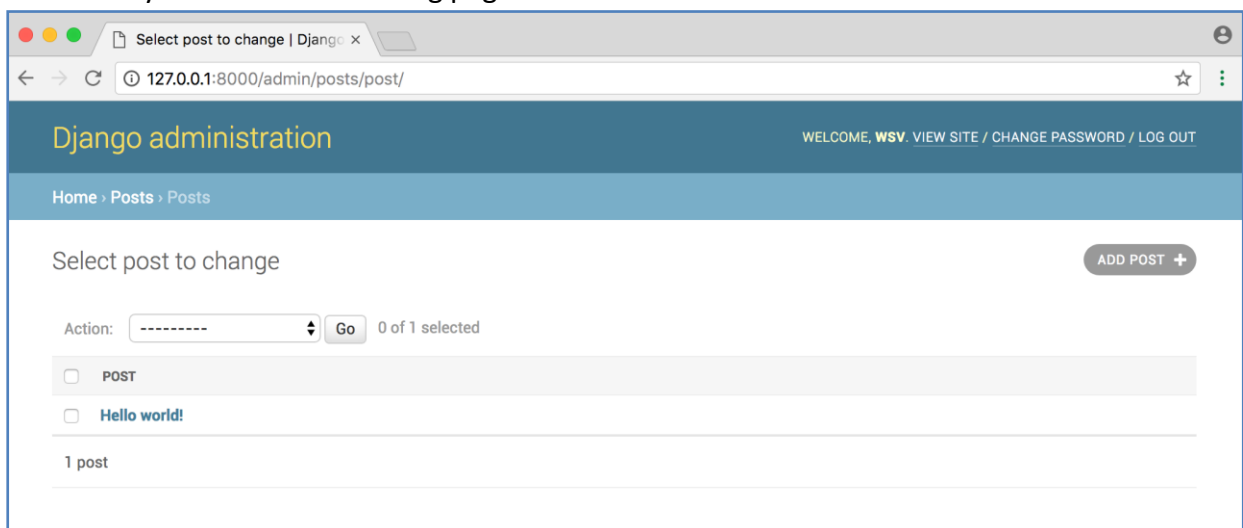




Click on the link for + Add next to Posts. Enter in the simple text Hello world!



On "save" you'll see the following page.



Now add our views file.

```
# posts/views.py
from django.views.generic import ListView
from .models import Post
class PostPageView(ListView):
    model = Post
    template_name = 'posts.html'
```

Create a posts.html template file.

```
(testy) D:\Python\testy> cd templates/posts.html
And add the code below to simply output all posts in the database.
<!-- templates/posts.html -->
<h1>Message board homepage</h1>
<ul>
    {% for post in object_list %}
        <li>{{ post.text }}</li>
    {% endfor %}
</ul>
```

Finally we need to update our urls.py files. Start with the project-level one.

```
# myproject/urls.py
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('', include('pages.urls')),
    path('admin/', admin.site.urls),
    path('posts/', include('posts.urls')),
]
```

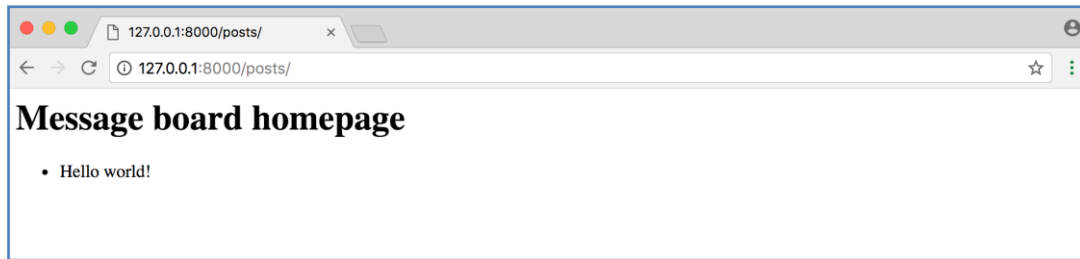
Then create a urls.py file in the posts app.

```
(testy) D:\Python\testy> cd posts/urls.py
```

And populate it as follows.

```
# posts/urls.py
from django.urls import path
from .views import PostPageView
urlpatterns = [
    path("", PostPageView.as_view(), name='posts'),
]
```

Okay, We're done. Start up the local server python manage.py runserver and navigate to our new message board page at <http://127.0.0.1:8000/posts>.



It simply displays our single post entry. Time for tests!

### TestCase

TestCase is the most common class for writing tests in Django. It allows us to mock queries to the database.

Let's test out our Post database model.

```
# posts/tests.py
from django.test import TestCase
from django.urls import reverse
from .models import Post
class PostTests(TestCase):
    def setUp(self):
        Post.objects.create(text='just a test')
    def test_text_content(self):
        post = Post.objects.get(id=1)
        expected_object_name = f'{post.text}'
        self.assertEqual(expected_object_name, 'just a test')
    def test_post_list_view(self):
        response = self.client.get(reverse('posts'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'just a test')
        self.assertTemplateUsed(response, 'posts.html')
```

With TestCase the Django test runner will create a sample test database just for our tests. Here we've populated it with the text 'just a test'.

In the first test we confirm that the test entry has the primary id of 1 and the content matches. Then in the second test on the view we confirm that that it uses the url name posts, has a 200 HTTP response status code, contains the correct text, and uses the correct template.

Run the new test to confirm everything works.

```
(testy) D:\Python\testy> python manage.py test
```

## Unit test

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. To achieve this, unittest supports some important concepts in an object-oriented way:

### test fixture

A *test fixture* represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

### test case

A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, `TestCase`, which may be used to create new test cases.

### test suite

A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

### test runner

A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

## Python's unittest2 library

**unittest2** is a backport of the new features added to the unittest testing framework in Python 2.7 and onwards. It is tested to run on Python 2.6, 2.7, 3.2, 3.3, 3.4 and pypy. To use unittest2 instead of unittest simply replace `import unittest` with `import unittest2`.

### New features include:

- **addCleanups** - better resource management
- *many* new assert methods including better defaults for comparing lists, sets, dicts unicode strings etc and the ability to specify new default methods for comparing specific types
- **assertRaises** as context manager, with access to the exception afterwards
- test discovery and new command line options (including failfast and better handling of ctrl-C during test runs)
- class and module level fixtures: **setUpClass**, **tearDownClass**, **setUpModule**, **tearDownModule**
- test skipping and expected failures
- new **delta** keyword argument to **assertAlmostEqual** for more useful comparison and for comparing non-numeric objects (like datetimes)
- **load\_tests** protocol for loading tests from modules or packages
- **startTestRun** and **stopTestRun** methods on `TestResult`

## Django Deployment to Github

Github is a global repository system which is used for version control. While working with django, if there is need for version management, it is recommended to use github. We will create and deploy a django project to the github so that it can be accessible globally.

Before deploying, **it is required to have a github account**, otherwise create an account first by visiting [github.com](https://github.com).

Open the terminal and **cd into the project**, we want to deploy. For example, our project name is djangoboot. Then

### Install Git

Goto this link : <https://git-scm.com/download/win>  
It will download Git.exe  
Run the installer and it will start wizard for installation press next on all steps.  
Then start CommandPrompt/

### Initialize Git

Use the following command to start the git.

```
D:\HRDK\Python\test\testDjango>git init  
Reinitialized existing Git repository in D:/HRDK/Python/test/testDjango/.git/
```

Provide global user name email for the project, it is only once, we don't need to provide it repeatedly.

```
D:\HRDK\Python\test\testDjango>git config --global user.name hardikchavda  
D:\HRDK\Python\test\testDjango>git config --global user.email hardikkchavda@gmail.com
```

### Create File

Create a file **.gitignore** inside the root folder of django project. And put the following code inside it.

```
// .gitignore  
*.pyc  
*~  
__pycache__  
myenv  
db.sqlite3  
/static  
.DS_Store
```

## Git Status

Check the git status by using the following command. It provides some detail to the screen.

```
D:\HRDK\Python\test\testDjango>git status
On branch master
Your branch is ahead of 'origin/master' by 8 commits.
(use "git push" to publish your local commits)

Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified: blog1/views.py
modified: templates/about.html

Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

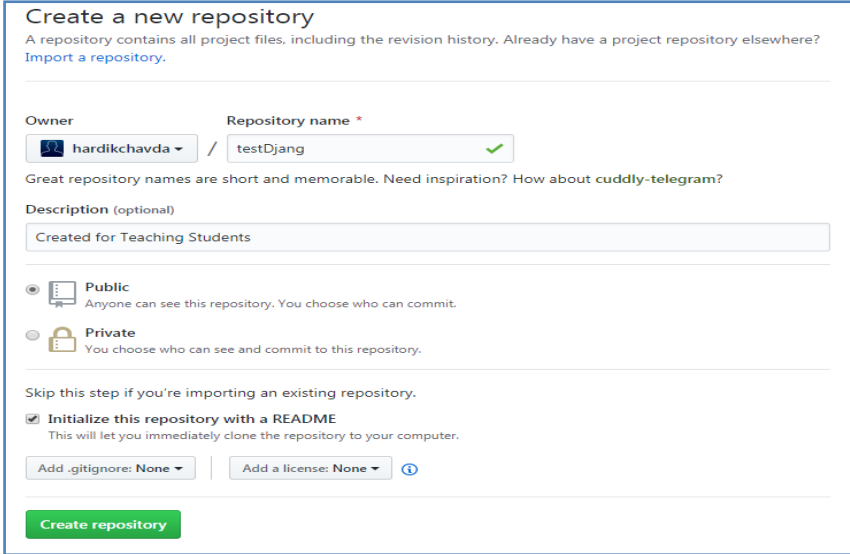
modified: blog1/urls.py
modified: blog1/views.py
modified: db.sqlite3
```

After saving, now execute the following command.

```
D:\HRDK\Python\test\testDjango>git add -all
D:\HRDK\Python\test\testDjango>git commit -m "my app first commit"
```

## Push to Github

First login into the git account and create a new repository and initialize with README. See the example.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?  
[Import a repository.](#)

Owner:  / Repository name:

Great repository names are short and memorable. Need inspiration? How about [cuddly-telegram?](#)

Description (optional)

Public  
Anyone can see this repository. You choose who can commit.

Private  
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

Initialize this repository with a README  
This will let you immediately clone the repository to your computer.

Add .gitignore:  | Add a license:  ⓘ

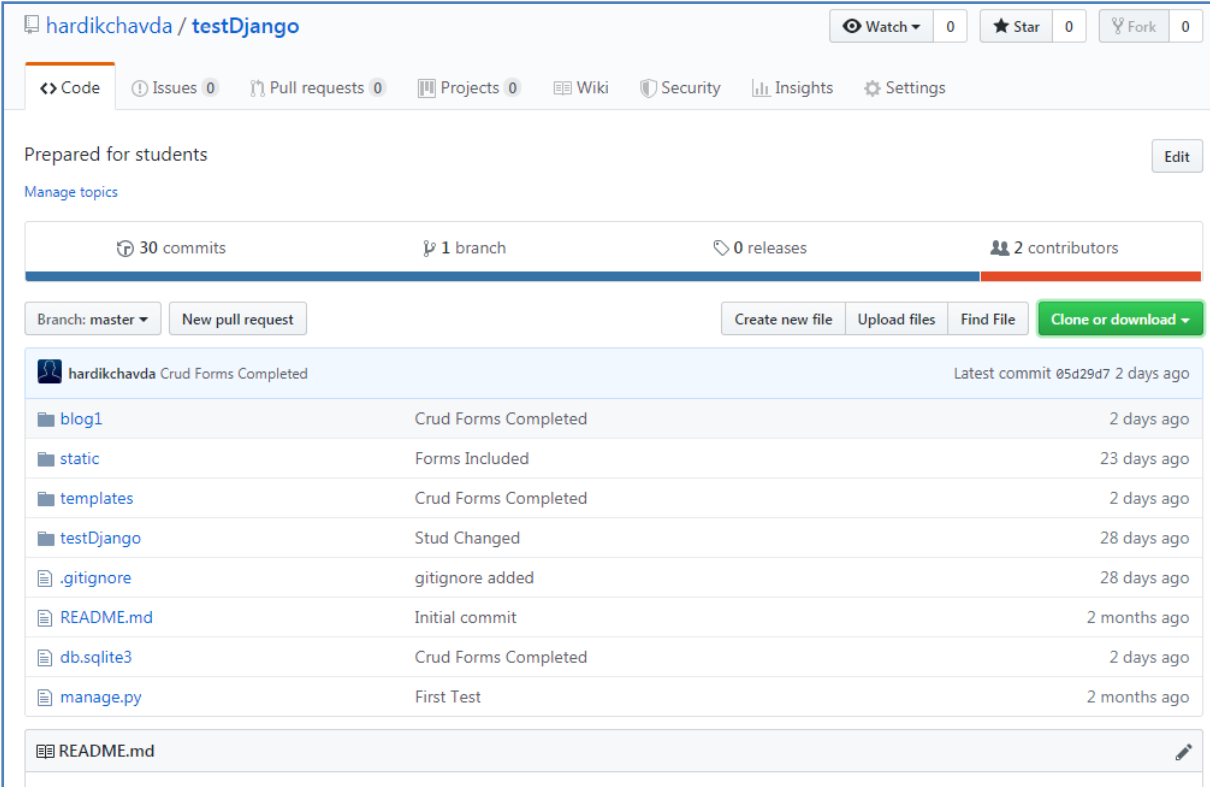


My repository name is testDjango. Click on the create repository button. Now repository has created. On next page, click on the clone button and copy the http url. In my case, it is <https://github.com/hardikchavda/testDjango.git>

Now, use this url with the following command.

```
D:\HRDK\Python\test\testDjango>git remote add origin https://github.com/hardikchavda/testDjango.git
D:\HRDK\Python\test\testDjango>git push -u --force origin master
```

Provide username and password of git account. It will start pushing project to the repository.



The screenshot shows the GitHub repository page for 'hardikchavda / testDjango'. The repository is prepared for students and has 30 commits, 1 branch, 0 releases, and 2 contributors. The main branch is 'master'. The repository contains several files and folders, including 'blog1', 'static', 'templates', 'testDjango', '.gitignore', 'README.md', 'db.sqlite3', and 'manage.py'. The latest commit is 'Crud Forms Completed' by hardikchavda, made 2 days ago.

File/Folder	Description	Time
hardikchavda	Crud Forms Completed	Latest commit 05d29d7 2 days ago
blog1	Crud Forms Completed	2 days ago
static	Forms Included	23 days ago
templates	Crud Forms Completed	2 days ago
testDjango	Stud Changed	28 days ago
.gitignore	gitignore added	28 days ago
README.md	Initial commit	2 months ago
db.sqlite3	Crud Forms Completed	2 days ago
manage.py	First Test	2 months ago

See, our django application has deploy successfully on github. Now, we can access it globally.