

# Diploma in IT, Networking and Cloud

## Module 4

# Python Programming and Django

## Theory Manual

Disclaimer: The content is curated for educational purposes only.

© Edunet Foundation. All rights reserved.

## Table of Contents

<b>Introduction to Python.....</b>	<b>14</b>
Overview of Python .....	14
Characteristics of Python .....	15
Advantages over other languages .....	16
<b>History of Python .....</b>	<b>19</b>
Python Timeline/History and IEEE rankings .....	19
<b>Features of Python .....</b>	<b>20</b>
Easy to Code and Understand .....	20
Expressive Language .....	21
Free and Open Source .....	22
Interpreted Language .....	22
Object-Oriented Language .....	23
Dynamically Typed Programming Language .....	23
Cross-Platform Language .....	24
Large Standard Library .....	24
Extensible Language .....	25
<b>Setting up Path .....</b>	<b>25</b>

Python Installation for Windows .....	25
How to set Python Path in Windows .....	27
Python Installation for Linux .....	28
How to set Python Path in Linux .....	28
<b>Basic Syntax Variable .....</b>	<b>29</b>
Interactive Mode Programming .....	29
Script Mode Programming .....	30
Python Identifiers .....	31
Reserved Keywords .....	32
Lines and Indentation .....	32
Multi Line Statement .....	33
Quotation in Python .....	35
Comments in Python .....	35
Using Blank Lines .....	36
Input From the User .....	36
Multiple Statement on a Single Line .....	38
Multiple Statements Groups as Suites .....	38
<b>Data Types Operator .....</b>	<b>39</b>
Types of operators .....	40
<b>Conditional Statements .....</b>	<b>50</b>

Statement and Description.....	50
Single Statement Suites .....	56
<b>Looping.....</b>	<b>57</b>
While Loop in Python.....	57
For Loop in Python.....	59
Nested Loop in Python .....	61
<b>Control Statements.....</b>	<b>63</b>
Break Statement.....	63
Continue Statement .....	65
Pass Statement .....	66
<b>String Manipulation .....</b>	<b>68</b>
Introduction.....	68
Create Strings .....	69
Index and Slice Strings.....	70
String Operators .....	72
String Formatting Operators .....	74
Common Python String Methods.....	79
Regular Expressions / Misc String functions .....	84
<b>List.....</b>	<b>90</b>
Create a list in Python.....	90

---

How to access elements from a list?.....	94
Slice lists in Python .....	95
Change or add elements to a list .....	96
Delete or Remove elements from a list.....	98
Python List Methods .....	100
List Comprehension: Elegant way to create new List .....	101
Other List Operations in Python.....	102
<b>Tuples.....</b>	<b>103</b>
Creating a Tuple .....	103
Advantages of Tuple over List.....	104
Access Tuple Elements.....	105
Performing Operations - Modifying, Deleting Python Tuple .....	107
Tuple Methods .....	108
Other Tuple Operations.....	109
<b>Functions and Methods.....</b>	<b>110</b>
What Is a Function in Python? .....	110
Call a Function .....	112
Function Examples.....	126
Functions as Objects .....	127
Function Attributes .....	128

Dictionary.....	129
Create a dictionary .....	129
Access Items in a Dictionary .....	130
Change or Add elements in a dictionary.....	130
Delete or remove elements from a dictionary .....	131
Python Dictionary Methods.....	132
Dictionary Comprehension.....	133
Other Dictionary Operations .....	133
Built-in Functions with Dictionary .....	134
<b>Functions.....</b>	<b>135</b>
Introduction / Overview.....	135
What is lambda in Python? .....	135
<b>Modules .....</b>	<b>142</b>
What are modules in Python? .....	142
Python Module: Mechanism .....	143
import modules in Python? .....	143
<b>Package .....</b>	<b>149</b>
What are packages? .....	149
Importing module from a package .....	150
<b>Input /Output.....</b>	<b>152</b>

---

Python Input, Output and Import.....	152
Output formatting.....	153
Python Input.....	154
What is a file?.....	154
How to open a file? .....	155
How to close a file? .....	159
Perform Write operation .....	161
Perform Read operation.....	162
Renaming and deleting files in Python .....	163
Python File object methods .....	164
<b>Exception Handling .....</b>	<b>174</b>
Exception Handling: Error vs. Exception .....	174
Handle Exceptions with Try-Except.....	175
Handling All Types of Exceptions with Except.....	176
Handling Multiple Exceptions with Except.....	177
Handle Exceptions with Try-Finally .....	177
Raise Exception with Arguments .....	178
Create Custom Exceptions .....	179
<b>Object Oriented Python.....</b>	<b>183</b>
Principles of object oriented programming.....	183

---

Class in Python.....	186
Constructors.....	190
Databases.....	193
<b>Python Web Django and Flask .....</b>	<b>205</b>
Django.....	205
FLASK.....	207
<b>Python for Web-Django .....</b>	<b>209</b>
Web Framework, Django Introduction, Django Architecture:.....	210
Model-View-Controller (MVC) Architecture .....	214
Installation of Django .....	216
<b>Writing your first Django app: basic poll application (Part1).....</b>	<b>223</b>
Creating a project .....	223
The development server.....	226
Creating the Polls app .....	228
Write your first view.....	229
path() argument: route.....	232
path() argument: view .....	232
path() argument: kwargs .....	232
path() argument: name .....	232
<b>Writing your first Django app, part 2 .....</b>	<b>234</b>



---

Database setup.....	234
Creating models.....	236
Activating models .....	238
Playing with the API .....	243
<b>Introducing the Django Admin .....</b>	<b>251</b>
Creating an admin user .....	252
Start the development server .....	252
Enter the admin site .....	253
Make the poll app modifiable in the admin.....	254
Explore the free admin functionality .....	254
<b>Request and response objects- .....</b>	<b>260</b>
Quick overview.....	260
HttpRequest objects.....	260
Attributes.....	260
Attributes set by application code .....	265
Attributes set by middleware .....	266
Methods .....	267
HttpResponse objects- .....	272
Usage .....	273
Attributes.....	275

---

Methods .....	276
HttpResponse subclasses .....	281
JsonResponse objects .....	283
Usage .....	284
<b>Working with forms .....</b>	<b>285</b>
HTML forms.....	286
GET and POST .....	287
Django's role in forms.....	287
Forms in Django .....	288
The Django Form class.....	288
Instantiating, processing, and rendering forms .....	289
Building a form .....	290
The work that needs to be done .....	290
Building a form in Django.....	291
More about Django Form classes.....	295
Bound and unbound form instances .....	295
More on fields.....	295
Working with form templates .....	298
Form rendering options .....	298
Rendering fields manually.....	299

Looping over the form's fields .....	302
<b>SQL operations in Django.....</b>	<b>312</b>
<b>Making queries.....</b>	<b>312</b>
Creating objects.....	314
Saving changes to objects .....	315
Saving ForeignKey and ManyToManyField fields.....	315
Retrieving objects.....	317
Retrieving all objects.....	317
Retrieving specific objects with filters- .....	318
Retrieving a single object with get()- .....	320
Other QuerySet methods- .....	321
Limiting QuerySets- .....	321
<b>Handling sessions .....</b>	<b>323</b>
Enabling sessions .....	323
Configuring the session engine- .....	323
Using database-backed sessions-.....	323
Using cached sessions- .....	324
Using file-based sessions-.....	325
Using cookie-based sessions- .....	325
Using sessions in views- .....	326



---

Working with JSON and AJAX.....	331
References .....	336

---

In this section, we will read about:

- Introduction to Python.
- History.
- Features
- Setting Up Path
- Basic Syntax Variable
- Data Types Operator
- Conditional Statement
- Looping
- Control Statement
- String Manipulation
- Lists
- Tuple
- Functions and Methods
- Dictionaries
- Functions
- Modules
- Input and Output
- Exception Handling

# Introduction to Python

## Overview of Python

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactic constructions than other languages.

- Python is Interpreted – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Python is Interactive – You can actually sit at a Python prompt and interact with the Interpreter directly to write your programs.
- Python is a Beginner's Language – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

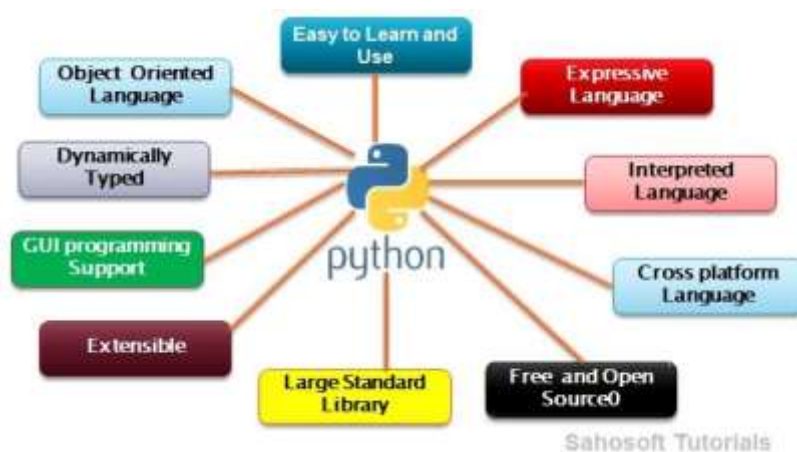


Image 1: Python- Overview of Python.

Reference: <https://i.pinimg.com/originals/ea/fa/99/eafa991b59dde12f94e41f3622d157b2.png>

## Characteristics of Python

As we discussed the introduction to python now we are going to learn about the characteristics of python are as below:

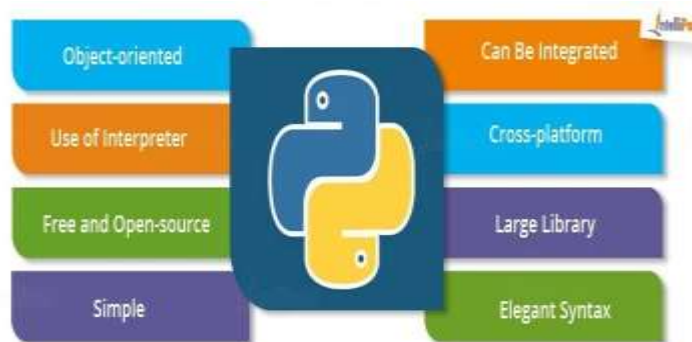


Image 2: Python - Characteristics of Python.

Reference: <https://intellipaat.com/mediaFiles/2019/02/TutorialPython-01.jpg>

### Platform Independent

Python is platform independent. The python code can be used for any operating system like Windows, Unix, Linux, and Mac. There is no need to write different code for the different OS.

### Interpreted

The python code does not need to compile as required for other languages. Python code automatically converts the source code into byte code internally and the code is

executed line by line not at once, so it takes more time to execute the code for the application.

## **Simple Syntax**

The Python language is simple and can be easily coded and read. The syntax of python is really simple and can be learned easily.

## **High Level Language**

It is a high-level language used for scripting. It means one does not need to remember the system architecture and no need for managing the memory as well.

## **Free and Open Source**

Python is Open Source and freely available over the internet anywhere. One does not need to take the license for it. It can be easily downloaded and use.

## **Rich Library Support**

Python can be integrated with other libraries that help in making the functionality work for you. Do not need to write the extra code for that.

## **Advantages over other languages**

One such language that is understood and preferred all over the world for development is Python. It's a general-purpose, high-level OOPs based interpreted language, used for dynamic applications, widely across the globe. Python is extremely popular for its versatility and scope of applicability.





Image 3: Python - advantages over other languages.

Reference: <https://i.redd.it/t0tjdyi6ewh21.jpg>

Python is a language embraced by a large community of coders for many reasons. Many businesses are advised to choose python as their language for programming. Let's find out why:

### **It is free**

Python is an open-source language. The Python Company is one of the largest companies and, still, is free. Using python language doesn't require a particular subscription or a custom-built platform either, thus any desktop and laptop is compatible with Python. All the tools that are necessary for python coding, the supporting means, modules, and libraries are absolutely free.

The essential IDEs that is, the integrated development environments that include PTVS, Pydevwith eclipse spyder python can be downloaded easily for free. Reduced costs are always beneficial to businesses.

---

## **It Needs Less Coding**

Python by nature has a very simple syntax. The same logic that needs 7 lines in a C++ language requires just 3 lines in Python. Having a smaller code requires less space, less time, and is well appreciated by coders, as the rework or correction also takes lesser time. The language takes all the parameters of readability. To support itself, the language has many built-ins and libraries that make it easy to comprehend.

## **All kinds of Businesses can afford it**

Being a free platform, all small and medium level companies can use it. Companies that are in the nascent stage can use the python platform and begin their operations with cost-effective software. The ability to develop applications and software quickly makes it suitable for startups, as they can survive in the cutthroat competition by leveraging the speed of the python language.

## **It is one of the most trending languages**

Java and C++ are the native languages with the Object-Oriented approach. Their use is very widespread, and efficiency is tremendous. The only problem with these languages is they are lengthy. Python, on the other hand, has all the features of object-oriented programming just like Java and C++, and is fast too. The codes are shorter and the syntax simple, thus being easy to amend, rework and optimize.

# History of Python

## Python Timeline/History and IEEE rankings

Python is a widely used general-purpose, high-level programming language.

- Python was conceptualized by Guido Van Rossum in the late 1980s.
- Rossum Published the first version of Python code (0.9.0) in February 1991 at CWI (Centrum Wiskunde & Informatica) in Netherland, Amsterdam.
- Python is Derived from the ABC Programming Language, which is a general-purpose Programming Language that has been developed at the CWI.
- Rossum chose the name “Python”, since he was a big fan of Monty Python’s Flying Circus.
- Python is now maintained by a core development team at the institute, although Rossum still holds a vital role in directing its progress.

The latest programming language ranking from IEEE spectrum maintains the same findings from last year’s report: Python is king for a majority of ranking metrics.

The report from IEEE Spectrum analyzes metrics from multiple data sources and ranks programming language popularity.

It ranks languages based on use cases:

Web development, enterprise, mobile, and embedded. Rankings also determine a language’s popularity based on different social channels, such as Twitter, StackOverflow, GitHub, and Reddit, and even ranks by which language is currently booming in the job market.



Image 4: Python- Feature: Easy to code and Understand.

Reference: <https://spectrum.ieee.org/image/MzExNjk1OA.jpeg>

# Features of Python

## Easy to Code and Understand

Python is high level programming language. Python is very easy to learn language as compared to other language like c, c#, java script, java etc. It is very easy to code in python language and anybody can learn python basic in a few hours or days. It is also a developer-friendly language.

## “Hello, World”

- **C**  

```
#include <stdio.h>

int main(int argc, char ** argv)
{
    printf("Hello, World!\n");
}
```
- **Java**  

```
public class Hello
{
    public static void main(String argv[])
    {
        System.out.println("Hello, World!");
    }
}
```
- **now in Python**  

```
print "Hello, World!"
```

Image 5: Python- Features: Easy to code and Understand

Reference: <https://analyticsprofile.com/wp-content/uploads/2018/08/python-1.jpg>

## Expressive Language

Python is very expressive when compared to other languages. By expressive, we mean, in Python a single line of code performs a lot more than what multiple lines can perform in other languages. In simple it means that fewer lines of code are required to write a program in Python.

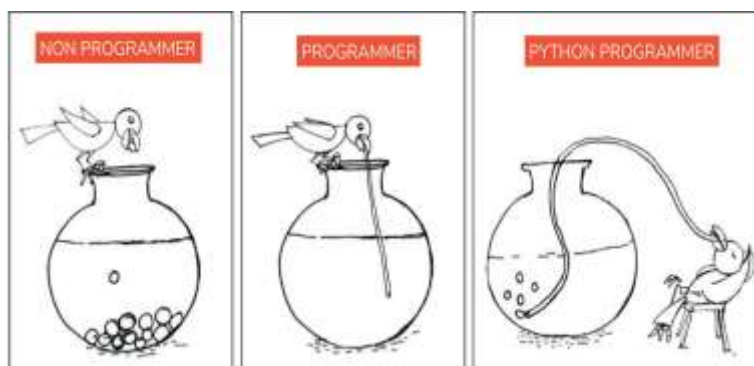


Image 6: Python-Feature: Expressive Language.

Reference: <https://i1.faceprep.in/Companies-1/features-of-python-language.png>

## Free and Open Source

Python is free and can be easily installed by anyone and on any system. Also, Python is an open-source programming language. This means that Python's source code can be freely modified and used by anyone.

## Interpreted Language

Python is an interpreted language. An interpreter in general works very differently from a compiler. An interpreter executes a code line by line and hence it gets easy for a programmer to debug errors. Also, if you have observed, **even though your program has multiple errors, Python displays only one error at a time**. Whereas a compiler compiles the entire code at once and displays a list of errors.

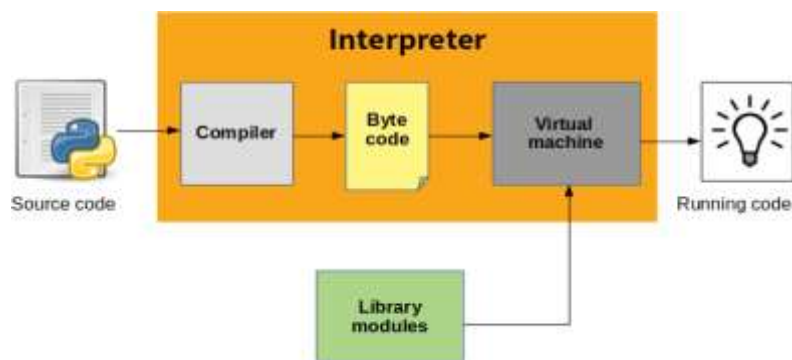


Image 7: Python-Feature: Interpreted Language.

Reference: <https://indianpythonista.files.wordpress.com/2018/01/pjme67t.png>

## Object-Oriented Language

Like other general-purpose languages, python is also an object-oriented language. In Python, **we can easily create and use classes and objects**. Some of the other major principles of object-oriented programming languages are Object, Class, Method, Inheritance, Polymorphism, Data Abstraction and Encapsulation.

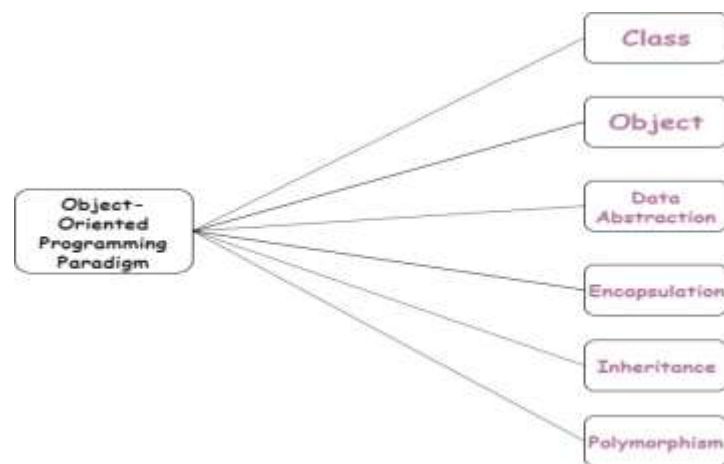


Image 8: Python-Feature:Object-Oriented Language.

Reference: <https://cdn.askpython.com/wp-content/uploads/2019/12/Python-oop.png>

## Dynamically Typed Programming Language

Python is a dynamically typed language. This means, whenever a variable is declared, the programmer need not mention its data type. Rather, the type of the variable is decided during runtime.

## Cross-Platform Language

Say, for example, you have written a piece of code in a Python IDE on Windows. Now, you want to run this code on another system. Then, you need not make any changes to the code to execute it on other machines like MAC, OS, Linux etc. The code remains exactly the same and this makes it easy for programmers to switch across platforms and work comfortably using Python.

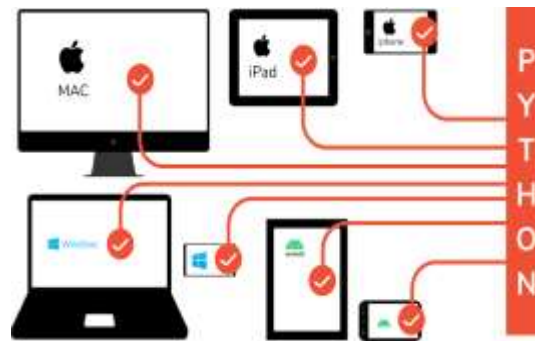


Image 9: Python-Feature: Cross-Platform Language.

Reference: <https://i1.faceprep.in/Companies-1/python-is-a-cross-platform-language.png>

## Large Standard Library

Python has a **large standard library** and this **helps save the programmers time** as you don't have to write your own code for every single logic. There are libraries for expressions, unit-testing, web browsers, databases, CGI, image manipulation etc.



## Extensible Language

In case you want to write a part of your Python code in C++ or Java etc, then you can do it. Since Python is an extensible language, it lets you do this with ease.

# Setting up Path

## Python Installation for Windows

List of Hardware/Software requirements:

1. Laptop/Computer with Windows/Linux OS - Ubuntu 18.04 LTS
2. Python Software Installation

Steps: Install Python software in the system.

- Open the browser and type the [python.org/downloads](https://python.org/downloads).
- Click on [download](#) .
- Choice either Windows x86-64 executable installer for 64-bit or Windows x86 executable installer for 32-bit.
- After downloading a file the below page will appear.



Image 10: Python- Installation of Python in windows.

Reference: <https://files.realpython.com/media/win-install-dialog.40e3ded144b0.png>



Image 11: Python-Installation of Python In windows.

Reference: <https://intellipaat.com/mediaFiles/2018/12/Step-4-1.png>

## How to set Python Path in Windows

To permanently modify the default **environment variables** :

My Computer > Properties > Advanced System Settings > Environment Variables > Edit

- Right-click 'My Computer'.
- Select 'Properties' at the bottom of the Context Menu.
- Select 'Advanced system settings'
- Click 'Environment Variables...' in the Advanced Tab
- Under 'System Variables': Click Edit

Add python's path to the end of the list (the paths are separated by semicolons(;))

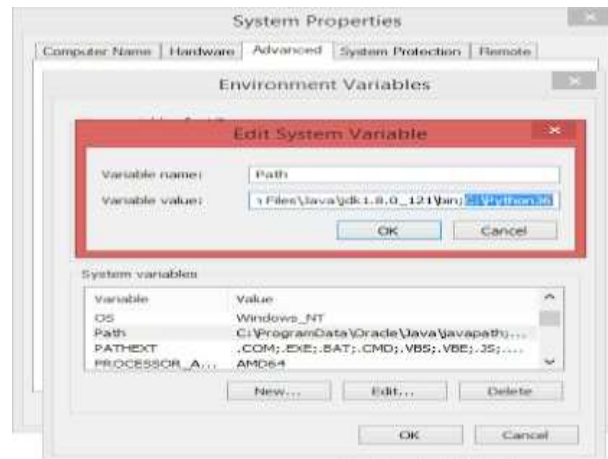


Image 12: Python-Setting up path in windows.

Reference: <https://windowsbulletin.com/wp-content/uploads/2019/08/Add-Path-to-Python-in-Windows.jpg>

## Python Installation for Linux

If you are using Ubuntu 16.10 or newer, then you can easily install Python 3.6 with the following commands:

```
$ sudo apt-get update  
$ sudo apt-get install python3.6
```

If you're using another version of Ubuntu (e.g. the latest LTS release), we recommend using the dead snakes PPA to install Python 3.6:

```
$ sudo apt-get install software-properties-common  
$ sudo add-apt-repository ppa:deadsnakes/ppa  
$ sudo apt-get update  
$ sudo apt-get install python3.6
```

## How to set Python Path in Linux

- Open your favorite terminal program;
- Open the file `~/.bashrc` in your text editor – e.g. `atom ~/.bashrc`;
- Add the following line to the end:

```
export PYTHONPATH=/home/my_user/code
```

Save the file.

- Close your terminal application;
- Start your terminal application again, to read in the new settings, and type this:

```
echo $PYTHONPATH
```

- It should show something like `/home/my_user/code`.

## Basic Syntax Variable

### Interactive Mode Programming

Interactive mode provides us with a quick way of running blocks or a single line of Python code. Interactive mode is handy when you just want to execute basic Python commands.

To access the Python shell, open the terminal of your operating system and then type "python". Press the enter key and the Python shell will appear.

```
C:\Windows\system32>python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` indicates that the Python shell is ready to execute and send your commands to the Python interpreter. The result is immediately displayed on the Python shell as soon as the Python interpreter interprets the command.

To run your Python statements, just type them and hit the enter key. You will get the results immediately, unlike in script mode. For example, to print the text "Hello World", we can type the following:

```
>>> print("Hello World")
Hello World
>>>
```

```
>>> 10
10
>>> print(5 * 20)
100
>>> "hi" * 5
```

Here are other examples:

```
'hihihihihi'
>>>
```

## Script Mode Programming

In script mode, You write your code in a text file then save it with a `.py` extension which stands for "Python". Note that you can use any text editor for this, including Sublime, Atom, notepad++, etc.

Let us create a new file from the Python shell and give it the name "hello.py". We need to run the "Hello World" program. Add the following code to the file:

```
print("Hello World")
```

Click "Run" then choose "Run Module". This will run the program:

## OUTPUT-

```
Hello World
```

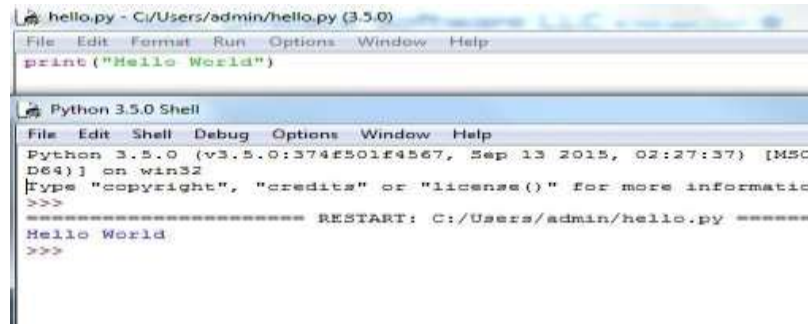


Image 13: Python- Script Mode Programming.

Reference: <https://s3.amazonaws.com/stackabuse/media/python-programming-interactive-vs-script-mode-1.jpg>

## Python Identifiers

An identifier is a name given to program elements such as variables, array, class and functions etc. An identifier is a sequence of letters, digits, and underscores, the first character of which cannot be a digit. Following rules must be kept in mind while naming an identifier.

- The first character must be an alphabet (uppercase or lowercase) or can be an underscore.
- An identifier can not start with a digit.
- All succeeding characters must be alphabet or digit.
- No special characters like !, @, #, \$, % or punctuation symbols is allowed except the underscore “\_”.
- No two successive underscores are allowed.
- Keywords can not be used as identifiers.

**Note-** Python is a case sensitive programming language, which means “ABC” and “abc” are not the same.

## Reserved Keywords

Python keywords are set words that cannot be used as an identifier in a program. These words are known as “Reserved Words” or “Keywords”. Python keywords are standard identifiers and their meaning and purpose is predefined by the compiler.

Below is a list of available keywords in Python Programming Language.

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

Image 14: Python- Script Mode Programming.

Reference: <https://makemeanalyst.com/wp-content/uploads/2017/06/keywords-python.png>

## Lines and Indentation

Python indentation is a way of telling the Python interpreter that a series of statements belong to a particular block of code. In languages like C, C++, Java, we use curly braces { } to indicate the start and end of a block of code. In Python, we use space/tab as indentation to indicate the same to the compiler.

- Python requires indentation as part of the syntax.
- Indentation signifies the start and end of a block of code.
- Program will not run without correct indentation.



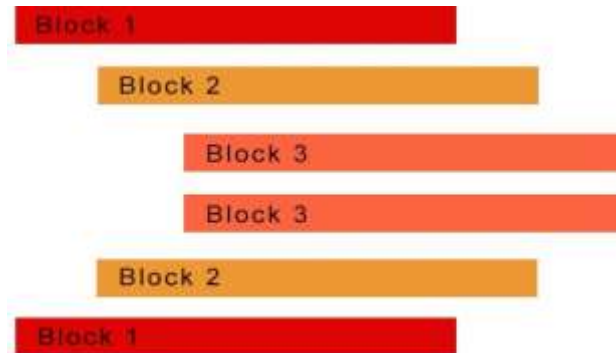


Image 15: Python- Lines and Indentation.

Reference: <https://www.stechies.com/userfiles/images/code-block.jpg>

```
a = input('Enter the easiest programming language: ') //statement 1
if a == 'python': //statement 2
    print('Yes! You are right') //statement 3
else: //statement 4
    print('Nope! You are wrong') //statement 5
```

In the above code,

Statement 1 gets executed first, then it gets to statement 2. Now only if statement 2 is correct, we would want statement 3 to get printed. Hence Statement 3 is indented with 2 spaces. Also, statement 5 should be printed, if statement 4 is true and hence this is also indented with 2 spaces.

Statement 1, 2 and 4 are main statements which need to be checked and hence these 3 are indented with the same space.

## Multi Line Statement

Usually, every Python statement ends with a newline character. However, we can extend it over to multiple lines using the line continuation character (`\`).

When you right away use the line continuation character (`\`) to split a statement into

multiplelines.

```
# Initializing a list using the multi-line statement
>>> my_list = [1, \
... 2, 3\
... ,4,5 \
... ]
>>> print(my_list)

[1, 2, 3, 4, 5]
```

```
>>> subjects = [
... 'Maths',
... 'English',
... 'Science'
... ]
>>> print(subjects)

['Maths', 'English', 'Science']
```

```
>>> type(subjects)

<class 'list'>
```

## Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal –

```
word = 'word' sentence = "This is a sentence." paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

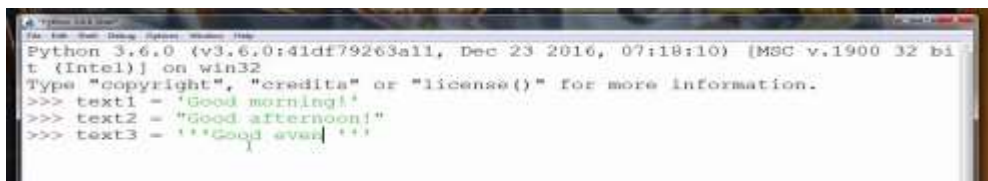


Image 16 : Python: Quotation in Python.

Reference: <https://i.ytimg.com/vi/uSief1mVRoY/maxresdefault.jpg>

## Comments in Python

A comment in Python starts with the hash character, # , and extends to the end of the physical line. A hash character within a string value is not seen as a comment, though. To be precise, a comment can be written in three ways - entirely on its own line, next to a statement of code, and as a multi-line comment block.

Single-line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.

**For Example-**

```
#This would be a comment in Python
```

Comments that span multiple lines – used to explain things in more detail – are created by adding a delimiter (""") on each end of the comment.

```
"""  
This would be a multiline comment  
in Python that spans several lines and  
describes your code, your day, or anything you want it to  
"""
```

## Using Blank Lines

Using blank lines in functions, sparingly, to indicate logical sections. Python accepts the control-L (i.e. ^L) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file.

- A line containing only whitespaces, possibly with a comment, is known as a blank line and Python totally ignores it.
- In an interactive interpreter session, you must enter a physical line to terminate a multiple statement.

## Input From the User

Python provides us with two inbuilt functions to read the input from the keyboard.

- `raw_input ( prompt )`
- `input ( prompt )`

**raw\_input ( )** : This function works in older versions (like Python 2.x). This function takes exactly what is typed from the keyboard, converts it to string and then returns it to the variable in which we want to store. For example –

```
# Python program showing  
# a use of raw_input()  
  
g = raw_input("Enter your name : ")  
print g
```

#### OUTPUT-

```
Enter your name : Priyanka Singh  
Priyanka Singh  
>>> |
```

**input ( )** : This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether the user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python. For example –

```
# Python program showing  
# a use of raw_input()  
  
val = input("Enter your value : ")  
print(val)
```

#### OUTPUT-

```
Enter your value : 1234  
1234  
>>> |
```

## Multiple Statement on a Single Line

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon –

```
a=10  
b=20  
c=a*b  
print (c)
```

### OUTPUT-

```
a=10; b=20; c=1*b; print (c)
```

## Multiple Statements Groups as Suites

A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines which make up the suite.

```
if expression :
```

```
    suite
```

```
elif expression :
```

```
    suite
```

```
else :
```

```
    suite
```

## Data Types Operator

In Python, operators are special symbols that designate that some sort of computation should be performed. The values that an operator acts on are called operands.

Here is an example:

```
>>> a = 10
>>> b = 20
>>> a + b
30
```

In this case, the + operator adds the operands a and b together. An operand can be either a literal value or a variable that references an object:

```
>>> a = 10
>>> b = 20
>>> a + b - 5
25
```

A sequence of operands and operators, like `a + b - 5`, is called an expression. Python supports many operators for combining data objects into expressions. These are explored below.

## Types of operators

- Python Arithmetic Operator
- Python Comparison Operator
- Python Assignment Operator
- Python Bitwise Operator
- Python Logical Operator
- Python Membership Operator
- Python Identity Operator
- Python Operator Precedence

### Python Arithmetic Operators

Arithmetic Operators perform various arithmetic calculations like addition, subtraction, multiplication, division, %modulus, exponent, etc. There are various methods for arithmetic calculation in Python like you can use the eval function, declare variable & calculate, or call functions.

Operator	Name	Description
+	Addition	The operands on either side of this operator are added.
-	Subtraction	The operand on the right side of this operator is subtracted from the one on the left side.
*	Multiplication	The operands on either side of this operator are multiplied



/	Division	The operand on the left side of this operator is divided by the one on the right side. As a result, it returns the quotient in the form of a floating-point value.
%	Modulo	The operand on the left side of this operator is divided by the one on the right side. As a result, it returns the remainder value.
**	Exponent	This operator raises the left side operand to the power of the right side operand.
//	Floor division	The operand on the left side of this operator is divided by the one on the right side. As a result, it returns the quotient in the form of an integer value.

## Python Comparison Operator

These operators compare the values on either side of the operand and determine the relation between them. It is also referred to as relational operators. Various comparison operators are ( ==, !=, <, >, <=, etc)

Operator	Name	Description
==	Equal to	Checks whether two operands are equal
!=	Not equal to	Checks whether two operands are not equal
>	Greater than	Checks whether the left side operand is greater than

		the rightside operand
<	Less than	Checks whether the left side operand is less than the rightside operand
>=	Greater than orequal to	Checks whether the left side operand is either greater orequal to the right side operand
<=	Lesser than orequal to	Checks whether the left side operand is either less than orequal to the right side operand

## Python Assignment Operator

Python assignment operators are used for assigning the value of the right operand to the leftoperand. Various assignment operators used in Python are (+=, -=, \*=, /=, etc.)

Operator	Name	Description
=	Assignment	This operator assigns its right-side value to its left-sideoperand
+=	Addition assignment	This operator adds left and the right side operands and assigns the result to the left side operand
-=	Subtraction assignment	This operator subtracts the right operand from the leftoperand and assigns the result to the left side operand

<code>*=</code>	Multiplication assignment	This operator multiplies the right-side operand with the left-side operand and assigns the result to the left side operand
<code>/=</code>	Division assignment	This operator divides the left side operand by the right-side operand and assigns the quotient to the left side operand
<code>%=</code>	Modulus assignment	This operator divides the left side operand by the right side operand and assigns the remainder to the left side operand
<code>**=</code>	Exponentiation assignment	This operator raises the left side operand to the power of the right-side operand and assigns the result value to the left side operand
<code>//=</code>	Floor division assignment	This operator divides the left operand by the right operand and assigns the quotient value (in the form of an integer value) to the left operand
<code>&amp;=</code>	Bitwise AND assignment	This operator performs a bitwise AND operation on both the left and the right side operands and assigns the result to the left side operand
<code> =</code>	Bitwise OR assignment	This operator performs a bitwise OR operation on both the left and the right side operands and assigns the result to the left side operand
<code>^=</code>	Bitwise XOR assignment	This operator performs bitwise XOR operation on both the left and the right side operands and assigns the result to the left side operand
<code>&gt;&gt;=</code>	Bitwise right shift	This operator right shifts the given value by the specified position and assigns the result to the left

	assignment	side operand
<<=	Bitwise left shift assignment	This operator left shifts the given value by the specified position and assigns the result to the left side operand

## Python Bitwise Operator

Bitwise operator works on bits and performs bit by bit operation. Assume if  $a = 60$ ; and  $b = 13$ ; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands –

$a = 0011\ 1100$

$b = 0000\ 1101$

-----

$a \& b = 0000\ 1100$

$a | b = 0011\ 1101$

$a \wedge b = 0011\ 0001$

$\sim a = 1100\ 0011$

There are following Bitwise operators supported by Python language

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

## Python Logical Operator

There are **three logical operators in Python** - and, or and not. They are used to combine two or more conditional statements.

Operator	Description
and	The output is true when both the expressions are true
or	The output is true if either one of the expression is true
not	Reverses the output

## Python Membership Operator

Membership operators **in** and **not in** are used to find whether a value is present in a particular Python object or not.

Operator	Description
in	Returns True if it finds a variable in the specified sequence and false otherwise.
not in	Returns False if it does not find a variable in the specified sequence and false otherwise.

## Python Identity Operator

Identity operators `is` and `is not` are used to check whether the memory locations of two variables/objects are the same or not. Also, we can use these operators to find if a variable/object belongs to a particular type or not.

Operator	Description
<code>is</code>	Returns True if the operands on either side of the operator point to the same object and false otherwise.
<code>is not</code>	Returns False if the operands on either side of the operator point to the same object and true otherwise.

## Python Operator Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description
1	<p><code>**</code></p> <p>Exponentiation (raise to the power)</p>

2	<p>~ + -</p> <p>Complement, unary plus and minus (method names for the last two are +@ and -@)</p>
3	<p>* / % //</p> <p>Multiply, divide, modulo and floor division</p>
4	<p>+ -</p> <p>Addition and subtraction</p>
5	<p>&gt;&gt; &lt;&lt;</p> <p>Right and left bitwise shift</p>
6	<p>&amp;</p> <p>Bitwise 'AND'</p>
7	<p>^  </p> <p>Bitwise exclusive 'OR' and regular 'OR'</p>



8	<p>&lt;= &lt; &gt; &gt;=</p> <p>Comparison operators</p>
9	<p>&lt;&gt; == !=</p> <p>Equality operators</p>
10	<p>= %= /= //= -= += *= **=</p> <p>Assignment operators</p>
11	<p>is is not</p> <p>Identity operators</p>
12	<p>in not in</p> <p>Membership operators</p>
13	<p>not or and</p> <p>Logical operators</p>

# Conditional Statements

## Statement and Description

Decision-making statements are those that **help in deciding the flow of the program**. For example, at times, you might want to execute a block of code only if a particular condition is satisfied. Well, in this case, a decision-making statement will be of great help. To understand this, let's now look at how they function.

### If statement

This is the simplest decision-making statement in Python. It is **used to decide if a particular block of code needs to be executed or not**. Whenever the given condition is **True**, then the block of code inside it gets executed, else it does not.

#### The If statement flowchart:

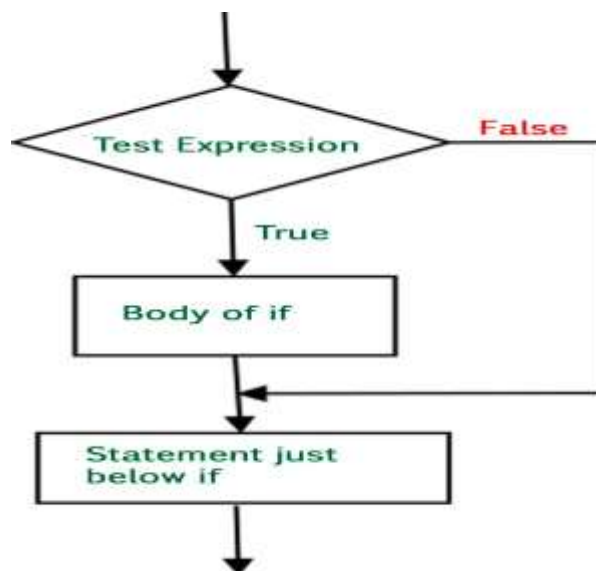


Image 17: Python – if statement in python

Reference: <https://tutorialspoint.dev/image/if-statement.jpg>

## Syntax for if statement-

```
if condition:  
    body of if
```

## Example of if statement

```
#program to check if num1 is less than num 2  
num1, num2 = 5, 6  
if(num1 < num2):  
    print("num1 is less than num2")  
Output:num1 is less than num2
```

**Explanation-**Here the value 5 is less than 6, so this means the condition is True. Hence the printstatement inside the body of if gets executed.

## If else statement

The statements written within the else block get executed whenever the if condition is **False**. You can imagine the flow of execution this way,

The if else statement flowchart:

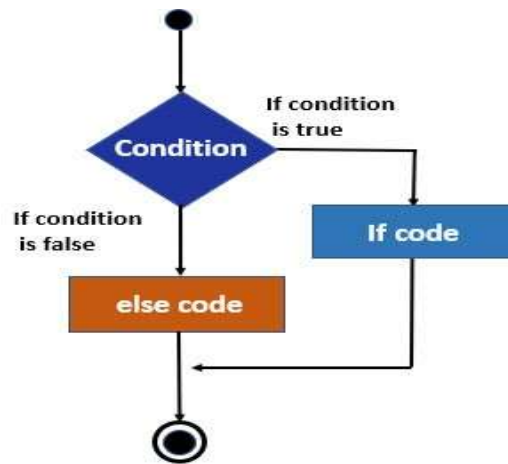


Image 18: Python -If else statement in Python.

Reference: <https://cdn.educba.com/academy/wp-content/uploads/2020/01/if-else-Statement-in-C-Flow.jpg>

### Syntax for if else statements

```
if condition:  
    body of if  
else:  
    body of else
```

### Example of if else statement-

```
#program to check if a num1 is less than num2
num1, num2 = 6, 5
if (num1 < num2):

    print("num1 is less than num2")
else:
    print("num2 is less than num1")
Output:num2 is lesser than num1
```

**Explanation-**In the above-given code, the if condition is False and hence the control shifts to the else block. Hence the statement written within the else block gets printed.

### Elif statement

The keyword **elif** is a combination of else and if. This statement works similar to 'else if' statements in the C and other languages. The statements in elif block get executed when the previous if condition is **False**.

### The elif statement flowchart:

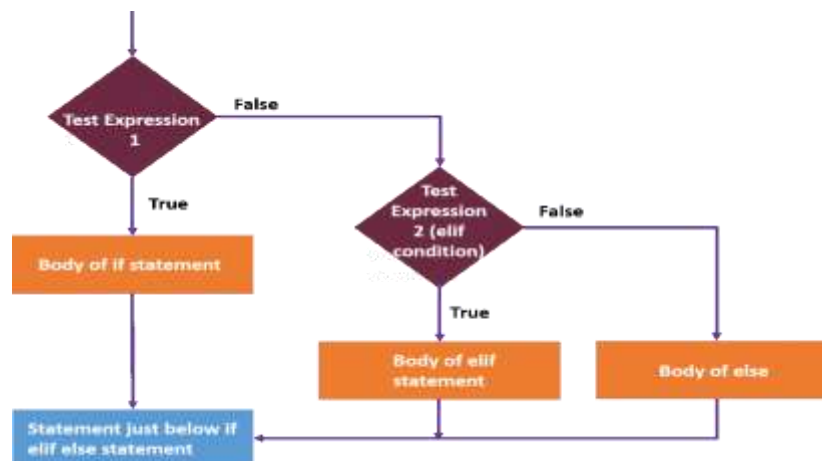


Image 19:Python - Elif Statement in Python.

Reference:<https://intellipaat.com/mediaFiles/2019/02/python14.png>

---

## Syntax for elif statements

```
if condition:  
    body of if  
elif condition:  
    body of elif  
else:  
    body of else
```

## Example of elif statement-

```
num1, num2 = 5, 5  
  
if(num1 > num2):  
    print("num1 is greater than num2")  
  
elif(num1 == num2):
```

```
    print("num1 is equal to num2")  
else:  
    print("num1 is less than num2")  
  
Output:num1 is equal to num2
```

**Explanation:** Since both the values are equal, the if condition is False. Now the control shifts to elif statement and here the condition is True. Hence the statements in the elif block get executed.

## Nested if statement

One if statement inside another if statement is called Nested if statement. The structure and flow of execution can be assumed as shown below.

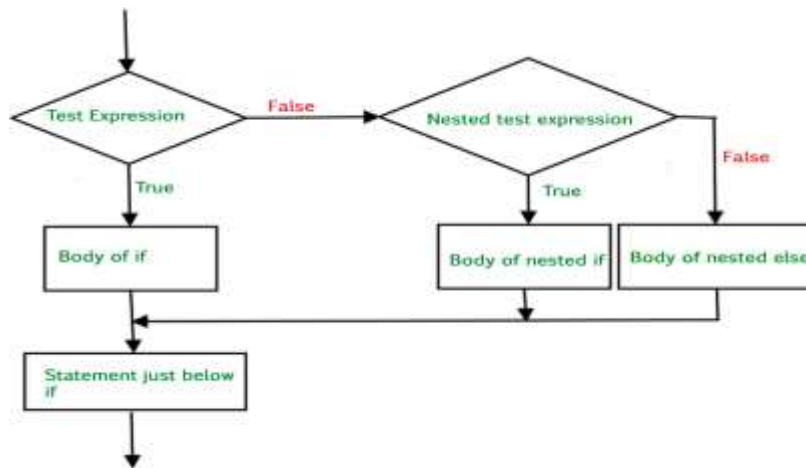


Image 20: Python - Nested if Statement in Python.

Reference: <https://media.geeksforgeeks.org/wp-content/uploads/nested-if.jpg>

## Syntax for nested if statements

```
if condition 1:  
    if condition 2:  
        statement  
    else:  
        statement  
else:  
    statement
```

### Example of nested if statement-

```
num1 = 5
if (num1 != 0):
    if(num1 > 0):
        print("num1 is a positive number")
    else:
        print("num1 is a negative number")
else:
    print("num1 is neither positive nor negative")
Output:num1 is a positive number
```

**Explanation:** Here, the first if condition is True. Hence the interpreter checks for the inner if condition. This is also True. Hence, the statements inside inner if get executed.

### Single Statement Suites

If the suite of an if clause consists only of a single line, it may go on the same line as the headerstatement.

Here is an example of a one-line if clause:



```
#!/usr/bin/python

var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

## OUTPUT-

```
Value of expression is 100
Good bye!
```

# Looping

Loops in Python programming function similar to loops in C, C++, Java or other languages. Python loops are used to repeatedly execute a block of statements until a given condition returns to be False. In Python, we have **3 different kinds of looping statements**, namely:

## While Loop in Python

While loop **iterates through a block of statements repetitively until the given condition returns False**. This means, while loop first checks the given condition and if the condition is **True**, then it executes the block of statements inside the while loop. Whenever the condition returns **False**, it breaks out of the loop and executes the next immediate line of code.

We use a **while loop** when we don't know the number of times to **iterate** through the block of statements.

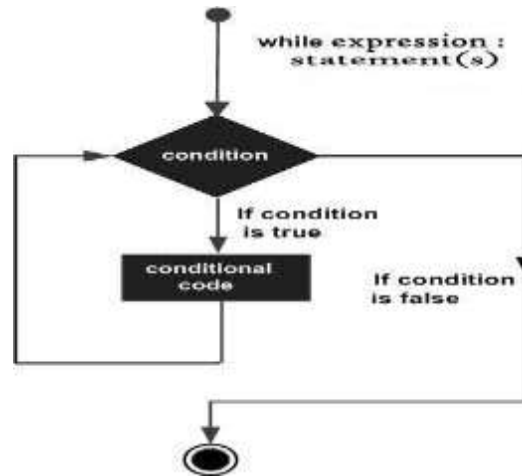


Image 21: Python - While Loop in Python.

Reference: [https://www.tutorialspoint.com/python/images/python\\_while\\_loop.jpg](https://www.tutorialspoint.com/python/images/python_while_loop.jpg)

Syntax for while loop in Python:  
while expression:  
    statement(s)

**Example:** Let's say you want to calculate the sum of all numbers less than a given number (n). This is how a while loop would help

```
number = 7
sum = 0
i = 0
while (i < number):
    sum = sum + i
    i = i + 1
print (sum)
```

Output: 21

**Explanation:** Here, the while loop executes until  $i = 6$ , when  $i = 7$ , the condition  $i < \text{number}$  returns **False**. Hence it breaks out of the while loop and executes the immediate next line of code i.e the print statement.

## For Loop in Python

The **for loop in Python** is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

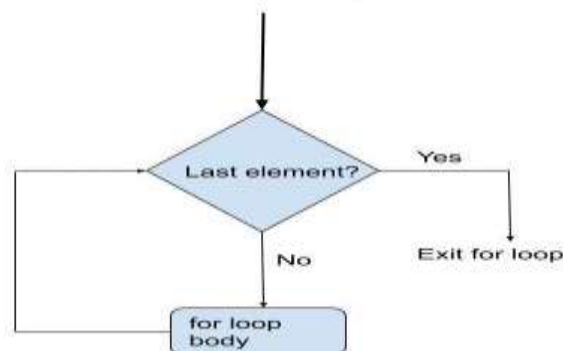


Image 22: Python - For Loop in Python.

Reference: <https://cdn.askpython.com/wp-content/uploads/2019/07/for-loop-flow-diagram.jpg>

## Syntax of for loop-

```
for iterating_var in sequence:  
    statements(s)
```

## Example-

```
even_numbers = [2, 4, 6, 8, 10]  
for i in even_numbers:  
    print(even_numbers)
```

Output:

```
[2, 4, 6, 8, 10]  
[2, 4, 6, 8, 10]  
[2, 4, 6, 8, 10]  
[2, 4, 6, 8, 10]  
[2, 4, 6, 8, 10]
```

**Example:** Now, let's say you want to print all the even numbers from 2 to 10. In this case, range function can be used this was

```
for i in range (2, 12, 2):  
    print (i)
```

Output:

```
2  
4  
6  
8  
10
```

**Explanation:** Here range function is used to iterate through the sequence 2 to 12 with a difference of 2 between the numbers. The syntax of range function used here is a range (lower limit, upper limit, the difference between numbers). In this case, the lower limit is inclusive and upper limit if exclusive of the sequence.

## Nested Loop in Python

**Loop inside a loop** is simply a nested loop. In Python, you can use any loop inside any other loop. For instance, a for in loop inside a while loop, a while inside for in and so on.

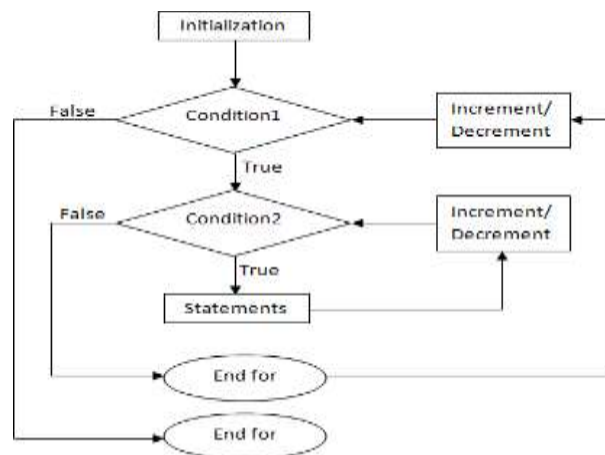


Image 23: Python - Nested Loop in Python.

Reference: <https://i.stack.imgur.com/OBYkj.png>

Syntax for for loop inside for loop (nested for loop):

for variable in sequence:

for variable in sequence:

statement (s)

statements(s)

Syntax for while loop inside while loop (nested while loop):

while condition:

while condition:

statement (s)

statements(s)

**Example of nested loops in Python:** Let's say you want to print the below pattern.

```
1
1 2
1 2 3
1 2 3 4
```

#program to print a pattern

for i in range (1, 5):

for j in range(i):

print (j + 1, end = ' ')

print ()

Output:

```
1
1 2
1 2 3
1 2 3 4
```

---

# Control Statements

Control statements in python are **used to control the flow of execution of the program based on the specified conditions**. Python supports **3 types of control statements** such as,

- 1) Break Statement
- 2) Continue Statement
- 3) Pass Statement

## Break Statement

The break statement in Python is used to terminate a loop. This means whenever the interpreter encounters the **break** keyword, **it simply exits out of the loop**. Once it breaks out of the loop, the control shifts to the immediate next statement.

Also, if the break statement is used inside a nested loop, it terminates the innermost loop and the control shifts to the next statement in the outer loop.

### Flowchart of Break Statement in Python:

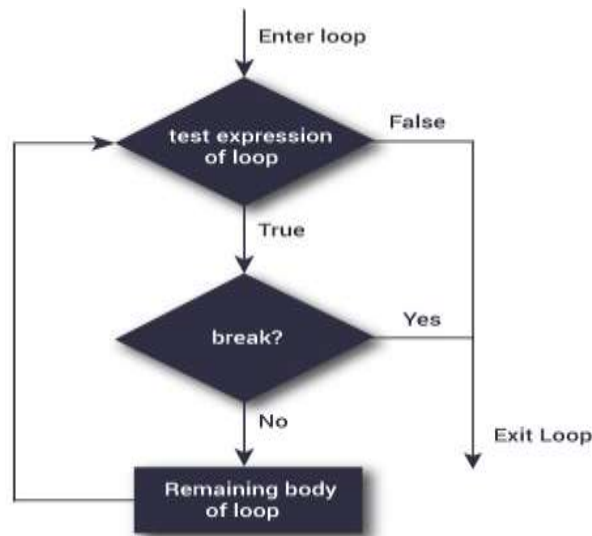


Image 24: Python - Break Statement in Python.

Reference: <https://cdn.programiz.com/sites/tutorial2program/files/flowchart-break-statement.jpg>

### Example of Break Statement in Python-

Here is an example of how break in Python can be used to exit out of the loop. Let's say you want to check if a given word contains the letter A, then how would you do it?

```
#program to check if letter 'A' is present in the input
a = input ("Enter a word")
for i in a:
    if (i == 'A'):
        print ("A is found")
        break
    else:
        print ("A not found")
```

Input: FACE Prep

Output:

A not found

A is found



How is break statement useful in the above example? Our code accesses each character of the string and checks if the character is 'A'. Whenever it finds character 'A', it breaks out of the loop, hence avoiding the process of checking the remaining letters. So here break helps avoid unwanted looping.

## Continue Statement

Whenever the interpreter encounters a continue statement in Python, **it will skip the execution of the rest of the statements in that loop and proceed with the next iteration**. This means it returns the control to the beginning of the loop. Unlike the break statement, continue statement does not terminate or exit out of the loop. Rather, it continues with the next iteration. Here is the flow of execution when a continue statement is used.

### Flowchart of Continue Statement in Python:

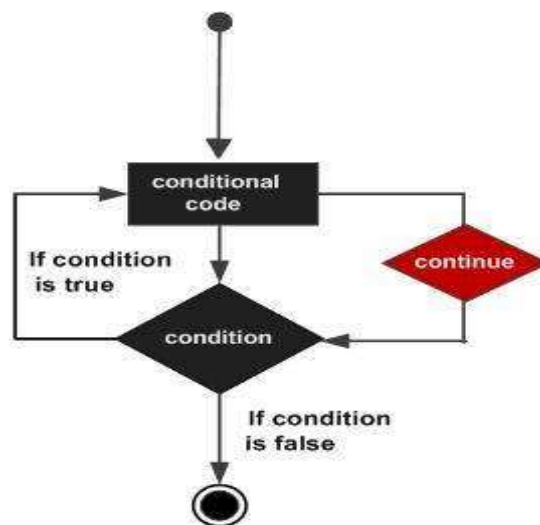


Image 25: Python - Continue Statement in Python.

Reference: [https://www.tutorialspoint.com/python/images/cpp\\_continue\\_statement.jpg](https://www.tutorialspoint.com/python/images/cpp_continue_statement.jpg)

---

### Example of continue statement-

Let us see how the same above discussed example can be written using a continue statement. Here is the code.

```
#program to check if letter 'A' is present in the input
a = input ("Enter a word")
for i in a:

    if (i != 'A'):

        continue
    else:

        print ("A is found")
```

Input: Edunet Foundation

Output:

A is found

## Pass Statement

The pass statement is used to execute nothing in Python. It is a null statement in Python. It is used as a placeholder. It can be used when we have a loop or function or class that have not yet been implemented.

### Flowchart of Pass Statement in Python:

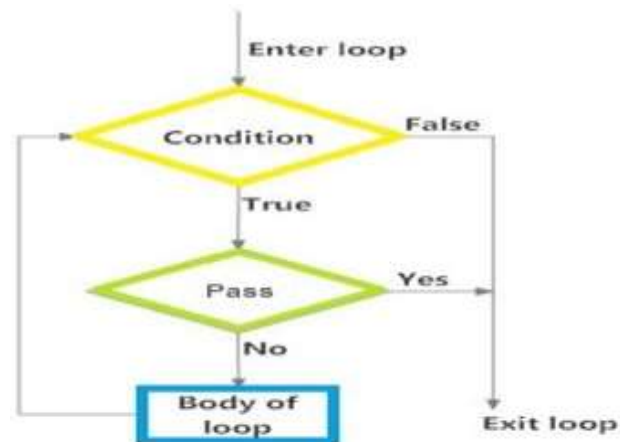


Image 26: Python - Pass Statement in Python.

Reference: <https://lh3.googleusercontent.com/proxy/Jpl8>

Assume we have a loop that is not implemented yet, but needs to be implemented in the future. In this case, if you leave the loop empty, the interpreter will throw an error. To avoid this, you can use the **pass** statement to construct a block that does nothing i.e contains no statements. Forexample,

```
for i in 'FIVE':  
    if (i == 'A'):  
        pass  
print (i)
```

Output:

```
F  
I  
V  
E
```

# String Manipulation

## Introduction

- Python string is an ordered collection of characters which is used to represent and store the text-based information.
- Strings are stored as individual characters in a contiguous memory location.
- It can be accessed from both directions: forward and backward.
- Characters are nothing but symbols.
- Strings are immutable, which means that once a string is created, they cannot be changed.

str = "HELLO"				
H	E	L	L	O
0	1	2	3	4
str[0] = 'H'	str[:] = 'HELLO'			
str[1] = 'E'	str[0:] = 'HELLO'			
str[2] = 'L'	str[:5] = 'HELLO'			
str[3] = 'L'	str[:3] = 'HEL'			
str[4] = 'O'	str[0:2] = 'HE'			
	str[1:4] = 'ELL'			

Image27: Python – String manipulation

Source: <https://static.javatpoint.com/python/images/strings-indexing-and-splitting2.png>

## Create Strings

- Creating strings is easy as you only need to enclose the characters either in single or double-quotes.
- In the following example, we are providing different ways to initialize strings.
- To share an important note that you can also use triple quotes to create strings. However, programmers use them to mark multi-line strings and docstrings

```
#Python string examples - all assignments are identical.
String_var = 'Python'
String_var = "Python"
String_var = """Python"""

# with Triple quotes Strings can extend to multiple lines
String_var = """ This document will help you to
explore all the concepts
of Python Strings!!! """

# Replace "document" with "tutorial" and store in another variable
substr_var = String_var.replace("document", "Tutorial")
print (substr_var)
```

When you run the program, the output will be:

```
>>>
= RESTART: C:/Users/Ragavan/AppData/Local/Programs/Python/Python37/StringDemo.py
  This Tutorial will help you to
  explore all the concepts
  of Python Strings!!!
>>>
```

## Index and Slice Strings

### Index a String in Python

P	Y	T	H	O	N	-	S	T	R	I	N	G
0	1	2	3	4	5	6	7	8	9	10	11	12
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Image 28: Python – Index string

Source: <https://cdn.techbeamers.com/wp-content/uploads/2016/03/String-Representation-in-Python.png>

- You need to know the index of a character to retrieve it from the String.
- Like most programming languages, Python allows to index from the zeroth position in Strings. But it also supports negative indexes. Index of '-1' represents the last character of the String. Similarly, using '-2', we can access the penultimate element of the string and so on.

```
sample_str = 'Python String'
print (sample_str[0])    # return 1st character
# output: P
print (sample_str[-1])   # return last character
# output: g
print (sample_str[-2])   # return last second character
# output: n
```

### Slice a String in Python

- To retrieve a range of characters in a String, we use 'slicing operator,' the colon ':' sign. With the slicing operator, we define the range as [a:b]. It'll let us print all the characters of the String starting from index 'a' up to char at index 'b-1'. So the char at index 'b' is not a part of the output.

```
sample_str = 'Python String'
print (sample_str[3:5])      #return a range of character
# ho
print (sample_str[7:])      # return all characters from index 7
# String
print (sample_str[:6])      # return all characters before index 6
# Python
print (sample_str[7:-4])
# St
```

## Modify/Delete

- Python Strings are by design immutable. It suggests that once a String binds to a variable, it can't be modified.
- If you want to update the String, then re-assign a new String value to the same variable.

```
sample_str = 'Python String'
sample_str[2] = 'a'

# TypeError: 'str' object does not support item assignment

sample_str = 'Programming String'
print (sample_str)

sample_str = "Python is the best scripting language."
del sample_str[1]
# TypeError: 'str' object doesn't support item deletion

del sample_str
print (sample_str)
# NameError: name 'sample_str' is not defined
```

- Similarly, we cannot modify the Strings by deleting some characters from it. Instead, we can remove the Strings altogether by using the 'del' command.

```
sample_str = "Python is the best scripting language."  
del sample_str[1]  
# TypeError: 'str' object doesn't support item deletion  
  
del sample_str  
print (sample_str)  
# NameError: name 'sample_str' is not defined
```

## String Operators

There are three types of operators supported by a string, which are:

- Basic Operators (+, \*)
- Relational Operators (<, >=<, >=, ==, !=)
- Membership Operators (in, not in)

**Concatenation (+)** - It combines two strings into one.

```
# example  
var1 = 'Python'  
var2 = 'String'  
print (var1+var2)  
# PythonString
```

**Repetition (\*)** - This operator creates a new string by repeating it a given number of times.

```
# example  
var1 = 'Python'  
print (var1*3)  
# PythonPythonPython
```



**Slicing [ ]** - The slice operator prints the character at a given index.

```
# example
var1 = 'Python'
print (var1[2])
# t
```

**Range Slicing [x: y]** - It prints the characters present in the given range.

```
# example
var1 = 'Python'
print (var1[2:5])
# tho
```

**Membership (in)** - This operator returns 'True' value if the character is present in the givenString.

```
# example
var1 = 'Python'
print ('n' in var1)
# True
```

**Membership (not in)** - It returns 'True' value if the character is not present in the given String.

```
# example
var1 = 'Python'
print ('N' not in var1)
# True
```

**Iterating (for)** - With this operator, we can iterate through all the characters of a string.

```
# example
for var in var1: print (var, end = "")
# Python
```

**Raw String (r/R)** - We can use it to ignore the actual meaning of Escape characters inside a string. For this, we add 'r' or 'R' in front of the String.

```
# example
print (r'\n')
# \n
print (R'\n')
# \n
```

## String Formatting Operators

### Python Escape Characters

- An Escape sequence starts with a backslash (\), which signals the compiler to treat it differently. Python subsystem automatically interprets an escape sequence irrespective of it is in a single-quoted or double-quoted Strings.
- Let's discuss an example-One of an important Escape sequence is to escape a single-quote or a double-quote.
- Suppose we have a string like – Python is a “widely” used language.
- The double-quote around the word “widely” disguise python that the String ends up there.
- We need a way to tell Python that the double-quotes inside the string are not the string markup quotes. Instead, they are the part of the String and should appear in the output.
- To resolve this issue, we can escape the double-quotes and single-quotes as:

```
print ("Python is a "widely" used language")
# SyntaxError: invalid syntax
# After escaping with double-quotes
print ("Python is a \"widely\" used language")
# Output: Python is a "widely" used language
```

## List of Escape Characters

- Here is the complete list of escape characters that are represented using backslash notation.

Escape Sequence	Description
\newline	Backslash and newline ignored
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\f	ASCII Formfeed
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab
\ooo	Character with octal value ooo
\xHH	Character with hexadecimal value HH

Image 29: python – Escape Characters

Source: <https://www.techbeamers.com/python-strings-functions>

Here are some examples

```
print("C:\\Python32\\Lib")
print("This is printed\nin two lines")
print("This is \\x48\\x45\\x58 representation")
```

## Output

```
C:\Python32\Lib
This is printed
in two lines
This is HEX representation
```

- Raw String to ignore escape sequence
- Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place `r` or `R` in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

```
print("This is \\x61 \\ngood example")
#Output : This is a
#Output : good example

print(r"This is \\x61 \\ngood example")
#Output : This is \\x61 \\ngood example
```

## Python Format Characters

- String `'%'` operator issued for formatting Strings. We often use this operator with the `print()` function.
- Here's a simple example.

```
print ("Employee Name: %s,\nEmployee Age:%d" % ('Alice',25))

# Employee Name: Alice,
# Employee Age: 25
```

## List of Format Symbols

Following is the table containing the complete list of symbols that you can use with the '%' operator.

Symbol	Conversion
%c	character
%s	string conversion via str() before formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPER-case letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPER-case 'E')
%f	floating-point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Image 30: Python – Format symbols

Source : <https://www.techbeamers.com/python-strings-functions-and-examples/#string-formatting-operators-in-python>

## The format() Method for Formatting Strings

- The `format()` method that is available with the string object is very versatile and powerful in formatting strings. Format strings contains curly braces `{}` as placeholders or replacement fields which gets replaced.

- We can use positional arguments or keyword arguments to specify the order.

```
# default(implicit) order
default_order = "{} , {} and {}".format('John', 'Bill', 'Sean')
print('\n--- Default Order ---')
print(default_order)

# order using positional argument
positional_order = "{1}, {0} and {2}".format('John', 'Bill', 'Sean')
print('\n--- Positional Order ---')
print(positional_order)

# order using keyword argument
keyword_order = "{s}, {b} and {j}".format(j='John', b='Bill', s='Sean')
print('\n--- Keyword Order ---')
print(keyword_order)
```

- The `format()` method can have optional format specifications. They are separated from field name using colon. For example, we can left-justify `<`, right-justify `>` or center `^` a string in the given space. We can also format integers as binary, hexadecimal etc. and floats can be rounded or displayed in the exponent format. There are a ton of formatting you can use.

```
# formatting integers
"Binary representation of {0} is {0:b}".format(12)

# formatting floats
"Exponent representation: {0:e}".format(1566.345)

# round off
"One third is: {0:.3f}".format(1/3)

# string alignment
"|{:<10}|{: ^10}|{:>10}|".format('butter', 'bread', 'ham')
```

## Old style formatting

- We can even format strings like the old `sprintf()` style used in C programming language. We use the `%` operator to accomplish this.

```
x = 12.3456789
print('The value of x is %3.2f' %x)
The value of x is 12.35
print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

## Common Python String Methods

### Unicode String support

- Regular Strings stores as the 8-bit ASCII value, whereas Unicode String follows the 16-bit ASCII standard. This extension allows the strings to include characters from the different languages of the world.
- In Python, the letter 'u' works as a prefix to distinguish between Unicode and usual strings.

```
print (u' Hello Python!!')

#Hello Python
```

### Built-in String Functions

- There are numerous methods available with the string object. The `format()` method that we mentioned above is one of them. Some of the commonly used methods are `lower()`, `upper()`, `join()`, `split()`, `find()`, `replace()` etc.

---

Here is a complete list of all the built-in methods to work with strings in Python.

<https://docs.python.org/3/library/string.html>

- Built-in String Functions Example

```
input_word = input('Enter your string') #Input given PyThONtHeOrYmODuLe
print(input_word.isupper()) #Output : False
print(input_word.upper()) #Output : PYTHONTHEORYMODULE
print(input_word.islower()) #Output : False
print(input_word.isupper()) #Output : False
print(input_word.capitalize()) #Output : pythontheorymodule
print(input_word.split()) #Output : ['PyThONtHeOrYmODuLe']
print(input_word.split(',')) #Output : ['PyThONtHeOrYmODuLe']
print(input_word.title()) #Output : Pythontheorymodule
print(input_word.strip()) #Output : PyThONtHeOrYmODuLe
```

## Conversion Function

- capitalize() – Returns the string with the first character capitalized and rest of the characters in lower case.
- lower() – Converts all the characters of the String to lowercase
- upper() – Converts all the characters of the String to uppercase
- swapcase() – Swaps the case of every character in the String means that lowercase characters got converted to uppercase and vice-versa.
- title() – Returns the ‘titlecased’ version of String, which means that all words start with uppercase and the rest of the characters in words are in lowercase.
- count( str[, beg [, end]]) – Returns the number of times substring ‘str’ occurs in the range [beg, end] if beg and end index are given else the search continues in full String Search is case-sensitive.

## Comparison Functions – Part

- islower() – Returns ‘True’ if all the characters in the String are in lowercase. If any of the char is in uppercase, it will return False.



- 
- `isupper()` – Returns 'True' if all the characters in the String are in uppercase. If any of the char is in lowercase, it will return False.
  - `isdecimal()` – Returns 'True' if all the characters in String are decimal. If any character in the String is of other data-type, it will return False.
  - `isdigit()` – Returns 'True' for any char for which `isdecimal()` would return 'True' and some characters in the 'No' category. If there are any characters other than these, it will return False'.
  - `isnumeric()` – Returns 'True' if all the characters of the Unicode String lie in any one of the categories Nd, No, and NI. If there are any characters other than these, it will return False.

Precisely, Numeric characters are those for which Unicode property includes: `Numeric_Type=Digit`, `Numeric_Type=Decimal` or `Numeric_Type=Numeric`.

- `isalpha()` – Returns 'True' if String contains at least one character (non-empty String), and all the characters are alphabetic, 'False' otherwise.
- `isalnum()` – Returns 'True' if String contains at least one character (non-empty String), and all the characters are either alphabetic or decimal digits, 'False' otherwise.

## Padding Functions

- `rjust(width[,fillchar])` – Returns string filled with input char while pushing the original content on the right side. By default, the padding uses a space. Otherwise, 'fillchar' specifies the filler character.
- `ljust(width[,fillchar])` – Returns a padded version of String with the original String left-justified to a total of width columns. By default, the padding uses a space. Otherwise, 'fillchar' specifies the filler character.
- `center(width[,fillchar])` – Returns string filled with the input char while pushing the original content into the center. By default, the padding uses a space. Otherwise, 'fillchar' specifies the filler character.

- `zfill(width)` – Returns string filled with the original content padded on the left with zeros so that the total length of String becomes equal to the input size. If there is a leading sign (+/-) present in the String, then with this function, padding starts after the symbol, not before it.

## Search Functions

- `find(str [,i [,j]])` – Searches for 'str' in complete String (if i and j not defined) or in a sub-string of String (if i and j are defined). This function returns the index if 'str' is found else returns '-1'. Here, i=search starts from this index, j=search ends at this index.
- `index(str[,i [,j]])` – This is same as 'find' method. The only difference is that it raises the 'ValueError' exception if 'str' doesn't exist.
- `rfind(str[,i [,j]])` – This is same as `find()` just that this function returns the last index where 'str' is found. If 'str' is not found, it returns '-1'.
- `count(str[,i [,j]])` – Returns the number of occurrences of substring 'str' in the String. Searches for 'str' in the complete String (if i and j not defined) or in a sub-string of String (if i and j are defined). Where: i=search starts from this index, j=search ends at this index. `var='This is a good example'`

## String Substitution Functions

- `replace(old,new[,count])` – Replaces all the occurrences of substring 'old' with 'new' in the String.
  - If the count is available, then only 'count' number of occurrences of 'old' will be replaced with the 'new' var.
  - Where old =substring to replace, new =substring
- `split([sep[,maxsplit]])` – Returns a list of substring obtained after splitting the String with 'sep' as a delimiter.

- Where, sep= delimiter, the default is space, maxsplit= number of splits to be done.
- splitlines(num) – Splits the String at line breaks and returns the list after removing the line breaks.
  - Where num = if this is a positive value. It indicates that line breaks will appear in the returned list.
- join(seq) – Returns a String obtained after concatenating the sequence 'seq' with a delimiter string.
  - Where: the seq= sequence of elements to join

## Misc String Functions

- lstrip([chars]) – Returns a string after removing the characters from the beginning of the String.
  - Where: Chars=this is the character to be trimmed from the String.
  - The default is whitespace character.
- rstrip() – Returns a string after removing the characters from the End of the String.
  - Where: Chars=this is the character to be trimmed from the String. The default is whitespace character.
- rindex(str[,i [,j]]) – Searches for 'str' in the complete String (if i and j not defined) or in a sub-string of String (if i and j are defined). This function returns the last index where 'str' is available.
  - If 'str' is not there, then it raises a ValueError exception.
  - Where: i=search starts from this index, j=search ends at this index.
- len(string) – Returns the length of given String

---

## Regular Expressions / Misc String functions

### What is Regular Expression?

- A regular expression in a programming language is a special text string used for describing a search pattern. It is extremely useful for extracting information from text such as code, files, log, spreadsheets or even documents.
- While using the regular expression the first thing is to recognize is that everything is essentially a character, and we are writing patterns to match a specific sequence of characters also referred as string. Ascii or latin letters are those that are on your keyboards and Unicode is used to match the foreign text.
- It includes digits and punctuation and all special characters like \$#@!%, etc.
- For instance, a regular expression could tell a program to search for specific text from the string and then to print out the result accordingly. Expression can include
  - Text matching
  - Repetition
  - Branching
  - Pattern-composition etc.

In Python, a regular expression is denoted as RE (REs, regexes or regex pattern) are imported through **re module**. Python supports regular expression through libraries. In Python regular expression supports various things like **Modifiers, Identifiers, and White space characters**.

Identifiers	Modifiers	White space characters	Escape required
\d= any number (a digit)	\d represents a digit. Ex: \d{1,5} it will declare digit between 1,5 like 424,444,545 etc.	\n = new line	. + * ? [] \$ ^ () {}   \
\D= anything but a number (anon-digit)	+ = matches 1 or more	\s= space	
\s = space (tab,space,newlineetc.)	? = matches 0 or 1	\t =tab	
\S= anything but a space	* = 0 or more	\e = escape	
\w = letters ( Match alphanumeric character, including "_")	\$ match end of a string	\r = carriage return	
\W =anything but letters ( Matches a non-alphanumeric character excluding "_")	^ match start of a string	\f= form feed	
. = anything but letters (periods)	matches either or x/y	-----	
\b = any character except for new line	[] = range or "variance"	-----	

\.	{x} = this amount of preceding code	-----	
----	-------------------------------------	-------	--

## Regular Expression Syntax - RE

```
import re
```

- "re" module included with Python primarily used for string searching and manipulation
- Also used frequently for web page "Scraping" (extract large amount of data from websites)
- We will begin the expression tutorial with this simple exercise by using the expressions(w+) and (^).
- Example of w+ and ^ Expression
  - "^": This expression matches the start of a string
  - "w+": This expression matches the alphanumeric character in the string
- Here we will see an example of how we can use w+ and ^ expression in our code. We cover re.findall function later in this tutorial but for a while we simply focus on \w+ and \^ expression.
- For example, for our string "python37, education is fun" if we execute the code with w+and^, it will give the output "python37".

```
import re
xx = "python37,education is fun"
r1 = re.findall(r"^\w+",xx)
print(r1)
```

- Remember, if you remove +sign from the w+, the output will change, and it will only give the first character of the first letter, i.e., [g]
- Example of \s expression in re.split function
- "s": This expression is used for creating a space in the string
- To understand how this regular expression works in Python, we begin with a simple example of a split function. In the example, we have split each word using the "re.split" function and at the same time we have used expression \s that allows to parse each word in the string separately.
- When you execute this code it will give you the output ['we', 'are', 'splitting', 'the', 'words'].
- Now, let see what happens if you remove "\" from s. There is no 's' alphabet in the output, this is because we have removed '\' from the string, and it evaluates "s" as a regular character and thus split the words wherever it finds "s" in the string.
- Similarly, there are series of other regular expressions in Python that you can use in various ways in Python like \d,\D,\$,\.,\b, etc.
- Here is the complete code

```
import re
xx = "python37,education is fun"
r1 = re.findall(r"^\w+", xx)
print((re.split(r'\s','we are splitting the words')))
print((re.split(r's','split the words')))
```

- Next, we will going to see the types of methods that are used with regular expressions.
- Using regular expression methods
- The "re" package provides several methods to actually perform queries on an input string.

The method we going to see -

- `re.match()`
- `re.search()`
- `re.findall()`

**Note:** Based on the regular expressions, Python offers two different primitive operations. The match method checks for a match only at the beginning of the string while search checks for a match anywhere in the string.

### Using `re.match()`

- The match function is used to match the RE pattern to string with optional flags. In this method, the expression "w+" and "\W" will match the words starting with letter 'g' and thereafter, anything which is not started with 'g' is not identified. To check match for each element in the list or string, we run the forloop.

### Finding Pattern in Text (`re.search()`)

- A regular expression is commonly used to search for a pattern in a text. This method takes a regular expression pattern and a string and searches for that pattern with the string.
- In order to use `search()` function, you need to import `re` first and then execute the code. The `search()` function takes the "pattern" and "text" to scan from our main string and returns a match object when the pattern is found or else not match if the pattern is not found.
- For example here we look for two literal strings "Software testing" "python37", in a text string "Software Testing is fun". For "software testing" we found the match hence it returns the output as "found a match", while for word "python37" we could not found in string hence it returns the output as "No match".



## re.findall()

- **findall()** module is used to search for “all” occurrences that match a given pattern. In contrast, **search()** module will only return the first occurrence that matches the specified pattern. **findall()** will iterate over all the lines of the file and will return all non-overlapping matches of pattern in a single step.
- For example, here we have a list of e-mail addresses, and we want all the e-mail addresses to be fetched out from the list, we use the **re.findall** method. It will find all the e-mail addresses from the list.
- Here is the complete code

```
import re
list = ["python37 get", "python37 give", "guru Selenium"]
for element in list:
    z = re.match("(g\\w+)\\W(g\\w+)", element)
    if z:
        print((z.groups()))

patterns = ['software testing', 'python37']
text = 'software testing is fun?'
for pattern in patterns:
    print('Looking for "%s" in "%s" ->' % (pattern, text), end=' ')
    if re.search(pattern, text):
        print('found a match!')
    else:
        print('no match')
abc = 'python37@google.com, careerpython37@hotmail.com, users@yahoo.com'
emails = re.findall(r'[\w\.-]+@[\w\.-]+', abc)
for email in emails:
    print(email)
```

# List

- The most commonly used data structure in Python is List. Python list is a container like an array that holds an ordered sequence of objects. The object can be anything from a string to a number or the data of any available type.
- A list can also be both homogenous as well as heterogeneous. It means we can store only integers or strings or both depending on the need. Next, every element rests at some position (i.e., index) in the list. The index can be used later to locate a particular item. The first indexes begin at zero, next are one, and so forth.

## Create a list in Python

- There are multiple ways to form a list in Python. Let's start with the most efficient one.
- Subscript operator
  - The square brackets [ ] represent the subscript operator in Python. It doesn't require a symbol lookup or a function call. Hence, it turns out the fastest way to create a list in Python.
- You can specify the elements inside [ ], separated by commas.
- Create a Python list using subscript operator

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed datatypes
my_list = [1, "Hello", 3.4]

# nested list - Also, a list can even have another list as an item.
# This is called nested list.
my_list = ["mouse", [8, 4, 6], ['a']]
```

## List() constructor

- Python includes a built-in list() method a.k.a constructor.
- It accepts either a sequence or tuple as the argument and converts into a Python list.
- Let's start with an example to create a list without any element.
- Create Python list using list()

```
# Create Python list using list()
# Syntax
theList = list([n1, n2, ...] or [n1, n2, [x1, x2, ...]])
```

- We can supply a standard or nested sequence as the input argument to the list() function. Let's first create an empty list.

```
#theList = list()
#empty list
len(theList)
```

- Note- The len() function returns the size of the list.

## List comprehension

- Python supports a concept known as "List Comprehension." It helps in constructing lists in an entirely natural and easy way.
- A list comprehension has the following syntax:

```
#Syntax - How to use List Comprehension
theList = [expression(iter) for iter in oldList if filter(iter)]
```

- It has square brackets grouping an expression followed by a for-in clause and zero or more if statements. The result will always be a list.

- Let's first see a simple example

```
theList = [iter for iter in range(5)]  
print(theList)
```

### Creating a multi-dimensional list

- You can create a sequence with a pre-defined size by specifying an initial value for each element.

```
init_list = [0]*3  
print(init_list)  
[0, 0, 0]
```

With the above concept, you can build a two-dimensional list.

```
two_dim_list = [ [0]*3 ] *3
```

The above statement works, but Python will only create the references as sublists instead of creating separate objects.

```
two_dim_list = [ [0]*3 ] *3
print(two_dim_list)
Output: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
two_dim_list[0][2] = 1
print(two_dim_list)
Output:[[0, 0, 1], [0, 0, 1], [0, 0, 1]]
```

We changed the value of the third item in the first row, but the same column in other rows also got affected.

### Extending a list

- Python allows lists to re-size in many ways. You can do that just by adding two or more of them.

```
L1 = ['a', 'b']
L2 = [1, 2]
L3 = ['Learn', 'Python']
L1 + L2 + L3
#Output : ['a', 'b', 1, 2, 'Learn', 'Python']
```

**List Extend ()** - Alternately, you can join lists using the extend() method.

```
L1 = ['a', 'b']
L2 = ['c', 'd']
L1.extend(L2)
print(L1)
#Output : ['a', 'b', 'c', 'd']
```

**List Append()** - Next, you can append a value to a list by calling the append() method. See the below example.

```
L1 = ['x', 'y']
L1.append(['a', 'b'])
L1
#Output : ['x', 'y', ['a', 'b']]
```

## How to access elements from a list?

There are various ways in which we can access the elements of a list.

### List Index

- We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.
- Nested list are accessed using nested indexing.

```
my_list = ['p','r','o','b','e']
# Output: p
print(my_list[0])

# Output: o
print(my_list[2])

# Output: e
print(my_list[4])

# Error! Only integer can be used for indexing
# my_list[4.0]

# Nested List
n_list = ["Happy", [2,0,1,5]]

# Nested indexing

# Output: a
print(n_list[0][1])

# Output: 5
print(n_list[1][3])
```

### Negative indexing

- Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_list = ['p', 'r', 'o', 'b', 'e']
```

```
# Output: e  
print(my_list[-1])
```

```
# Output: p  
print(my_list[-5])
```

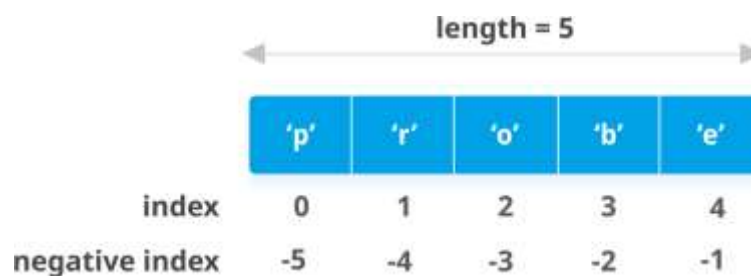


Image 31: Python – Negative Index

source: <https://cdn.programiz.com/sites/tutorial2program/files/python-list-index.png>

## Slice lists in Python

- We can access a range of items in a list by using the slicing operator (colon).

```
my_list = ['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']  
# elements 3rd to 5th  
print(my_list[2:5])
```

```
# elements beginning to 4th  
print(my_list[:-5])
```

```
# elements 6th to end  
print(my_list[5:])
```

```
# elements beginning to end  
print(my_list[:])
```

- Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.

## Change or add elements to a list

- List are mutable, meaning, their elements can be changed unlike string or tuple.
- We can use assignment operator (=) to change an item or a range of items.

```
# mistake values
odd = [2, 4, 6, 8]

# change the 1st item
odd[0] = 1

# Output: [1, 4, 6, 8]
print(odd)

# change 2nd to 4th items
odd[1:4] = [3, 5, 7]

# Output: [1, 3, 5, 7]
print(odd)
```

We can add one item to a list using `append()` method or add several items using `extend()` method.

```
odd = [1, 3, 5]

odd.append(7)

# Output: [1, 3, 5, 7]
print(odd)

odd.extend([9, 11, 13])

# Output: [1, 3, 5, 7, 9, 11, 13]
print(odd)
```



---

We can also use + operator to combine two lists. This is also called concatenation.

```
odd = [1, 3, 5]

# Output: [1, 3, 5, 9, 7, 5]
print(odd + [9, 7, 5])

#Output: ["re", "re", "re"]
print(["re"] * 3)
```

The \* operator repeats a list for the given number of times.

```
odd = [1, 9]
odd.insert(1,3)

# Output: [1, 3, 9]
print(odd)

odd[2:2] = [5, 7]

# Output: [1, 3, 5, 7, 9]
print(odd)
```

## Delete or Remove elements from a list

- We can delete one or more items from a list using the keyword `del`. It can even delete the list entirely.

```
my_list = ['p','r','o','b','l','e','m']

# delete one item
del my_list[2]

# Output: ['p', 'r', 'b', 'l', 'e', 'm']
print(my_list)

# delete multiple items
del my_list[1:5]

# Output: ['p', 'm']
print(my_list)

# delete entire list
del my_list

# Error: List not defined
print(my_list)
```

- We can use `remove()` method to remove the given item or `pop()` method to remove an item at the given index.
- The `pop()` method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure).
- We can also use the `clear()` method to empty a list.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
my_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']
print(my_list)

# Output: 'o'
print(my_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']
print(my_list)

# Output: 'm'
print(my_list.pop())

# Output: ['r', 'b', 'l', 'e']
print(my_list)

my_list.clear()

# Output: []
print(my_list)
```

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
my_list[2:3] = []
my_list
#Output:['p', 'r', 'b', 'l', 'e', 'm']
my_list[2:5] = []
my_list
#Output:['p', 'r', 'm']
```

---

## Python List Methods

- Methods that are available with list object in Python programming are tabulated below.
- They are accessed as `list.method()`. Some of the methods have already been used above.

### Python List Methods

- `append()` - Add an element to the end of the list
- `extend()` - Add all elements of a list to the another list
- `insert()` - Insert an item at the defined index
- `remove()` - Removes an item from the list
- `pop()` - Removes and returns an element at the given index
- `clear()` - Removes all items from the list
- `index()` - Returns the index of the first matched item
- `count()` - Returns the count of number of items passed as an argument
- `sort()` - Sort items in a list in ascending order
- `reverse()` - Reverse the order of items in the list
- `copy()` - Returns a shallow copy of the list

## Some examples of Python list methods

```
my_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1
print(my_list.index(8))

# Output: 2
print(my_list.count(8))

my_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]
print(my_list)

my_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]
print(my_list)
```

## List Comprehension: Elegant way to create new List

- List comprehension is an elegant and concise way to create a new list from an existing list in Python.
- List comprehension consists of an expression followed by for statement inside square brackets.
- Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]
# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
print(pow2)
```

This code is equivalent to

```
pow2 = []
for x in range(10):
    pow2.append(2 ** x)
```

- A list comprehension can optionally contain more for or if statements. An optional if statement can filter out items for the new list. Here are some examples.

```
pow2 = [2 ** x for x in range(10) if x > 5]
pow2
#Output: [64, 128, 256, 512]

odd = [x for x in range(20) if x % 2 == 1]
odd
#Output: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

[x+y for x in ['Python ', 'C '] for y in ['Language', 'Programming']]
#Output: ['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

## Other List Operations in Python

**List Membership Test** - We can test if an item exists in a list or not, using the keyword `in`.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True
print('p' in my_list)

# Output: False
print('a' in my_list)

# Output: True
print('c' not in my_list)
```

**Iterating Through a List** - Using a for loop we can iterate through each item in a list.

```
for fruit in ['apple', 'banana', 'mango']:
    print("I like", fruit)
```

# Tuples

- A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas, in a list, elements can be changed.

## Creating a Tuple

- A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.
- A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Empty tuple
my_tuple = ()
print(my_tuple) # Output: ()
# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple) # Output: (1, 2, 3)
# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple) # Output: (1, "Hello", 3.4)
# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
# Output: ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"
print(my_tuple)    # Output: 3, 4.6, "dog"
# tuple unpacking is also possible
a, b, c = my_tuple
print(a)           # 3
print(b)           # 4.6
print(c)           # dog
```

- Creating a tuple with one element is a bit tricky.
- Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("hello")
print(type(my_tuple)) # <class 'str'>
# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple)) # <class 'tuple'>
# Parentheses is optional
my_tuple = "hello",
print(type(my_tuple)) # <class 'tuple'>
```

## Advantages of Tuple over List

- Since tuples are quite similar to lists, both of them are used in similar situations as well.
- However, there are certain advantages of implementing a tuple over a list.

Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuples are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a



---

dictionary. With lists, this is not possible.

- o If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

## Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

### 1. Indexing

- We can use the index operator [] to access an item in a tuple where the index starts from 0.
- So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an element outside of tuple (for example, 6, 7,...) will raise an IndexError.
- The index must be an integer; so we cannot use float or other types. This will result in TypeError.
- Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[0])    # 'p'
print(my_tuple[5])    # 't'
# IndexError: list index out of range
# print(my_tuple[6])
# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]
# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
# nested index
print(n_tuple[0][3])    # 's'
print(n_tuple[1][1])    # 4
```

## 2. Negative Indexing

- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_tuple = ('p','e','r','m','i','t')
# Output: 't'
print(my_tuple[-1])
# Output: 'p'
print(my_tuple[-6])
```

## 3. Slicing

- We can access a range of items in a tuple by using the slicing operator - colon ":".

```
my_tuple = ('p','r','o','g','r','a','m','i','z')
# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])
# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:2])
# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])
# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.

## Performing Operations - Modifying, Deleting Python Tuple

### 1. Changing a Tuple

- Unlike lists, tuples are immutable.
- This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.
- We can also assign a tuple to different values (reassignment).

```
my_tuple = (4, 2, 3, [6, 5])
# TypeError: 'tuple' object does not support item assignment
# my_tuple[1] = 9
# However, item of mutable element can be changed
my_tuple[3][0] = 9      # Output: (4, 2, 3, [9, 5])
print(my_tuple)
# Tuples can be reassigned
my_tuple = ('p','r','o','g','r','a','m','i','z')
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple)
```

- We can use + operator to combine two tuples. This is also called concatenation.
- We can also repeat the elements in a tuple for a given number of times using the \* operator.
- Both + and \* operations result in a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))
# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

## 2. Deleting a Tuple

- We cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.
- But deleting a tuple entirely is possible using the keyword `del`.

```
my_tuple = ('p','r','o','g','r','a','m','i','z')
# can't delete items
# TypeError: 'tuple' object doesn't support item deletion
# del my_tuple[3]
# Can delete an entire tuple
del my_tuple
# NameError: name 'my_tuple' is not defined
print(my_tuple)
```

## Tuple Methods

- Methods that add items or remove items are not available with tuple. Only the following two methods are available.
- Python Tuple Methods
  - o **count(x)** Returns the number of items x
  - o **index(x)** Returns the index of the first item that is equal to x

Some examples of Python tuple methods:

```
my_tuple = ('a','p','p','l','e',)
print(my_tuple.count('p'))    # Output: 2
print(my_tuple.index('l'))    # Output: 3
```

## Other Tuple Operations

- Tuple Membership Test - We can test if an item exists in a tuple or not, using the keyword **in**.

```
my_tuple = ('a','p','p','l','e',)
# In operation
# Output: True
print('a' in my_tuple)
# Output: False
print('b' in my_tuple)
# Not in operation
# Output: True
print('g' not in my_tuple)
```

- Iterating Through a Tuple - Using a **for loop** we can iterate through each item in a tuple.

```
for name in ('John','Kate'):
    print("Hello",name)

# Output:
Hello John
Hello Kate
```

# Functions and Methods

## What Is a Function in Python?

- A function in Python is a logical unit of code containing a sequence of statements indented under a name given using the “def” keyword.
- Functions allow you to create a logical division of a big project into smaller modules. They make your code more manageable and extensible.
- Basically, there are three types of functions:
  - Python Built-in functions (an already created, or predefined, function)
  - User-defined function (a function created by users as per the requirements)
  - Anonymous function (a function having no name)

### Function in Python - Create & Def

#### Statement Create a Function – Syntax

```
#Single line function:
def single_line(): statement

#Python function with docstring:
def fn(arg1, arg2,...):
    """docstring"""
    statement1
    statement2

#Nested Python function:
def fn(arg1, arg2,...):
    """docstring"""
    statement1
    statement2
    def fn_new(arg1, arg2,...):
        statement1
        statement2
    ...
    ...
```

## Def Statement

- The “def” keyword is a statement for defining a function in Python.
- You start a function with the def keyword, specify a name followed by a colon (:) sign.
- The “def” call creates the function object and assigns it to the name given.
- You can further re-assign the same function object to other names.
- Give a unique name to your function and follow the same rules as naming the identifiers.
- Add a meaningful docstring to explain what the function does. However, it is an optional step.
- Now, start the function body by adding valid Python statements each indented with four spaces.
- You can also add a statement to return a value at the end of a function. However, this step is optional.
- Since def is a statement, so you can use it anywhere a statement can appear – such as nested in an if clause or within another function.
- Example :

```
if test:
    def test(): # First definition
    ...
else:
    def test(): # Alternate definition
    ...
    ...
```

## Call a Function

### Example of a Function Call & Polymorphism in Python How to Call a Function in Python?

- By using the def keyword, you learned to create the blueprint of a function which has a name, parameters to pass and a body with valid Python statements.
- You can do so by calling it from the Python script or inside a function or directly from the Python shell.
- To call a function, you need to specify the function name with relevant parameters, and that's it.
- Follow the below example to learn how to call a function in Python.
- Python function for modular

Programming Example of a Function Call:

It's a simple example where a function "typeOfNum()" has nested functions to decide on a number is either odd or even.

```
def typeOfNum(num): # Function header
# Function body
if num % 2 == 0:
def message():
print("You entered an even number.")
else:
def message():
print("You entered an odd number.")
message()
# End of function
typeOfNum(2) # call the function
typeOfNum(3) # call the function again
```



## Polymorphism in Python

- In Python, functions polymorphism is possible as we don't specify the argument types while creating functions.
- The behavior of a function may vary depending upon the arguments passed to it.
- The same function can accept arguments of different object types.
- If the objects find a matching interface, the function can process them.

### Example

```
def product(x, y) : return x * y
print(product(4, 9)) # function returns 36
print(product('Python!', 2)) # function returns
# Python!Python!
print(product('Python 2 or 3?', '3')) # TypeError occurs
```

The above example clarifies that we can pass any two objects to the product() function which supports the '\*' operator.

Some points which you should remember are as follows:

- Python is a dynamically typed language which means the types correlate with values, not with variables. Hence, the polymorphism runs unrestricted.
- That's one of the primary differences between Python and other statically typed languages such as C++ or Java.
- In Python, you don't have to mention the specific data types while coding.
- However, if you do, then the code limits to the types anticipated at the time of coding.
- Such code won't allow other compatible types that may require in the future.
- Python doesn't support any form of function overloading.

## How to Pass Parameters to a Function Parameters in a Function

- Parameters are the variables used in the function definition whereas arguments are the values we pass to the function parameters.
- Python supports different variations of passing parameters to a function.
- The argument gets assigned to the local variable name once passed to the function.
- Changing the value of an argument inside a function doesn't affect the caller.
- If the argument holds a mutable object, then changing it in a function impacts the caller.
- We call the passing of immutable arguments as Pass by Value because Python doesn't allow them to change in place.
- The passing of mutable arguments happens to be Pass by Pointer in Python because they are likely to be affected by the changes inside a function.

Example: Immutable vs. Mutable

```
def test1(a, b) :  
    a = 'Garbage' # 'a' receives an immutable object  
    b[0] = 'Python' # 'b' receives a list object  
    # list is mutable  
    # it can undergo an in place change  
def test2(a, b) :  
    a = 'Garbage 2'  
    b = 'Python 3' # 'b' now is made to refer to new  
    # object and therefore argument 'y'  
    # is not changed  
arg1 = 10  
arg2 = [1, 2, 3, 4]  
test1(arg1, arg2)  
print("After executing test 1 =>", arg1, arg2)  
test2(arg1, arg2)  
print("After executing test 2 =>", arg1, arg2)  
  
#After execution, the above code prints the following.  
  
After executing test 1 => 10 ['Python', 2, 3, 4]  
After executing test 2 => 10 ['Python', 2, 3, 4]
```

Example: How to avoid changing the mutable argument

```
def test1(a, b) :
    a = 'Garbage'
    b[0] = 'Python'
    arg1 = 10
    arg2 = [1, 2, 3, 4]
    print("Before test 1 =>", arg1, arg2)
    test1(arg1, arg2[:]) # Create an explicit copy of mutable object
    # 'y' in the function.
    # Now 'b' in test1() refers to a
    # different object which was initially a
    # copy of 'arg2'
    print("After test 1  =>", arg1, arg2)

#After execution, the above code prints the following.

Before test 1 => 10 [1, 2, 3, 4]
After test 1  => 10 [1, 2, 3, 4]
```

## Standard arguments

- The standard arguments are those which you pass as specified in a Python function definition. It means without changing their order and without skipping any of them.

```
def fn(value):
    print(value)
    return
fn()
```

- Executing the above code throws the below error as we've not passed the single argument required.
- `TypeError: fn() missing 1 required positional argument: 'value'`

---

## Keyword-based arguments

- When you assign a value to the parameter (such as param=value) and pass to the function (like fn(param=value)), then it turns into a keyword argument.
- If you pass the keyword arguments to a function, then Python determines it through the parameter name used in the assignment.
- See the below example.

```
def fn(value):  
    print(value)  
    return  
fn(value=123) # output => 123  
fn(value="Python!") # output => Python!
```

While using keyword arguments, you should make sure that the name in the assignment should match with the one in the function definition. Otherwise, Python throws the `TypeError` as shown below.

```
fn(value1="Python!") # wrong name used in the keyword argument
```

The above function call causes the following error.

`TypeError: fn() got an unexpected keyword argument`

### 'value1' Arguments with Default Values

- Python functions allow setting the default values for parameters in the function definition. We refer them as the default arguments.
- The callee uses these default values when the caller doesn't pass them in the function call.
- The below example will help you clearly understand the concept of default arguments.

```
def daysInYear(is_leap_year=False):  
    if not is_leap_year:  
        print("365 days")  
    else:  
        print("366 days")  
    return  
daysInYear()  
daysInYear(True)
```

Here, the parameter “is\_leap\_year” is working as a default argument. If you don’t pass anyvalue, then it assumes the default which is False.

The output of the above code is:

365 days

366 days

## Variable Arguments

- You may encounter situations when you have to pass additional arguments to a Pythonfunction. We refer them as variable-length arguments.
- The Python’s print() is itself an example of such a function which supports variablearguments.
- To define a function with variable arguments, you need to prefix the parameter with anasterisk (\*) sign. Follow the below syntax.

---

```
def fn([std_args,] *var_args_tuple ):
    """docstring"""
    function_body
    return_statement
    Check out the below example for better clarity.
    def inventory(category, *items):
    print("%s [items=%d]: " % (category, len(items)), items)
    for item in items:
    print("-", item)
    return
    inventory('Electronics', 'tv', 'lcd', 'ac', 'refrigerator', 'heater')
    inventory('Books', 'python', 'java', 'c', 'c++')
```

## Global, Local and Nonlocal

### variablesGlobal Variables

- In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

#### Example 1: Create a Global Variable

```
x = "global"
def foo():
    print("x inside:", x)
foo()
print("x outside:", x)

#Output be:
x inside: global
x outside: global
```

In the above code, we created x as a global variable and defined a foo() to print the global variable x. Finally, we call the foo() which will print the value of x.

What if you want to change the value of x inside a function?

```
x = "global"
def foo():
    x = x * 2
    print(x)
foo()

#Output be:

UnboundLocalError:
local variable 'x' referenced before assignment
```

The output shows an error because Python treats x as a local variable and x is also not defined inside foo().

## Local Variables

- A variable declared inside the function's body or in the local scope is known as localvariable.

Accessing local variable outside the scope

```
def foo():
    y = "local"
    foo()
    print(y)

#Output
NameError: name 'y' is not defined
```

Create a Local Variable

- Normally, we declare a variable inside the function to create a local variable.

```
def foo():
    y = "local"
    print(y)
foo()

#Output:
local
```

## Global and local variables

Here, we will show how to use global variables and local variables in the same code. Using Global and Local variables in the same code

```
x = "global"
def foo():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)
foo()

#Output
global global
local
```

- In the above code, we declare x as a global and y as a local variable in the foo(). Then, we use multiplication operator \* to modify the global variable x and we print both x and y.
- After calling the foo(), the value of x becomes global global because we used the x \* 2 to print two times global. After that, we print the value of local variable y i.e local.

Example: Global variable and Local variable with same name

```
x = 5
def foo():
    x = 10
    print("local x:", x)
foo()
print("global x:", x)

#Output
local x: 10
global x: 5
```



In the above code, we used the same name `x` for both global variable and local variable. We get a different result when we print the same variable because the variable is declared in both scopes, i.e. the local scope inside `foo()` and global scope outside `foo()`.

When we print the variable inside `foo()` it outputs local `x: 10`. This is called the local scope of the variable.

Similarly, when we print the variable outside the `foo()`, it outputs global `x: 5`. This is called the global scope of the variable.

### Nonlocal Variables

- Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.
- We use the `nonlocal` keyword to create a nonlocal variable.

Example: Create a nonlocal variable

```
def outer():
    x = "local"
    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)
        inner()
        print("outer:", x)
    outer()
#Output
inner: nonlocal
outer: nonlocal
```

In the above code, there is a nested function `inner()`. We use `nonlocal` keyword to create a nonlocal variable. The `inner()` function is defined in the scope of another function `outer()`.

**Note:** If we change value of nonlocal variable, the changes appear in the local variable.

## Global Keyword

### What is the global keyword

- In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.
- Rules of global Keyword
- The basic rules for global keyword in Python are:
  - When we create a variable inside a function, it is local by default.
  - When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
  - We use global keyword to read and write a global variable inside a function.
  - Use of global keyword outside a function has no effect.
  - Use of global Keyword

Example 1: Accessing global Variable from inside a Function

```
c = 1 # global variable
def add():
    print(c)
add()
```

### Example 2: Modifying Global Variable From Inside the Function

```
c = 1 # global variable
def add():
    c = c + 2 # increment c by 2
    print(c)
add()
```

### Example 3: Changing Global Variable from inside a Function using global

```
c = 0 # global variable
def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)
add()
print("In main:", c)
```

## Global in Nested Functions

- Here is how you can use a global variable in nested function.

### Example: Using a Global Variable in Nested Function

```
def foo():
    x = 20
    def bar():
        global x
        x = 25
    print("Before calling bar: ", x)
    print("Calling bar now")
    bar()
    print("After calling bar: ", x)
foo()
print("x in main: ", x)
```

## Name Resolution in a Python Function

- It is essential to understand how name resolution works in case of a def statement.
- Here are a few points you should keep in mind.
- The name assignments create or change local names.
- The LEGB rule comes in the picture for searching the name reference.
  - local – L
  - then enclosing functions (if any) – E
  - next comes the global – G
  - and the last one is the built-in – B

To gain more understanding, run through the below example:

```
#var = 5
def fn1():
    #var = [3, 5, 7, 9]
    def fn2():
        #var = (21, 31, 41)
        print(var)
    fn2()
    #uncomment var assignments one-by-one and check the output
fn1()
print(var)
#After uncommenting the first "var" assignment
#the output is:
5
5
#Next, after uncommenting the second "var"
#assignment as well, the output is:
[3, 5, 7, 9]
5
#Finally, if we uncomment the last "var" assignment,
#then the result is as follows.
(21, 31, 41)
5
```

## Scope Lookup in Functions

- Python functions can access names in all available enclosing def statements..

```
X = 101 # global scope name - unused
def fn1():
    X = 102 # Enclosing def local
    def fn2():
        print(X) # Reference made in nested def
    fn2() # Prints 102: enclosing def local
fn1()
```

The scope lookup remains in action even if the enclosing function has already returned.

```
def fn1():
    print('In fn1')
    X = 100
    def fn2():
        print('In fn2')
        print(X) # Remembers X in enclosing def scope
        return fn2 # Return fn2 but don't call it
    action = fn1() # Make, return function
    action() # Call fn2() now: prints 100
```

## Return Values

- In Python functions, you can add the “return” statement to return a value.
- Usually, the functions return a single value. But if required, Python allows returning multiple values by using the collection types such as using a tuple or list.
- This feature works like the call-by-reference by returning tuples and assigning the results back to the original argument names in the caller.

```
def returnDemo(val1, val2):
    val1 = 'Windows'
    val2 = 'OS X'
    # return multiple values in a tuple
    return val1, val2

var1 = 4
var2 = [2, 4, 6, 8]
print("before return =>", var1, var2)
var1, var2 = returnDemo(var1, var2)
print("after return =>", var1, var2)

#Output

before return => 4 [2, 4, 6, 8]
after return => Windows OS X
```

## Function Examples

- **General Function** - Check out a general function call example.

```
def getMin(*varArgs):
    min = varArgs[0]
    for i in varArgs[1:]:
        if i < min :
            min = i
    return min
min = getMin(21, -11, 17, -23, 6, 5, -89, 4, 9)
print(min)
#The output is as follows.
-89
```

- **Recursive Function** - Example of the recursive function.

```
def calcFact(num):  
    if(num != 1):  
        return num * calcFact(num-1)  
    else:  
        return 1  
print(calcFact(4))
```

## Functions as Objects

- Yes, Python treats everything as an object and functions are no different.
- You can assign a function object to any other names.

```
def testFunc(a, b) : print('testFunc called')  
fn = testFunc  
fn(22, 'bb')
```

You can even pass the function object to other functions.

```
def fn1(a, b) : print('fn1 called')  
def fn2(fun, x, y) : fun(x, y)  
fn2(fn1, 22, 'bb')
```

You can also embed a function object in data structures.

```
def fn1(a) : print('fn1', a)  
def fn2(a) : print('fn2', a)  
listOfFuncs = [(fn1, "First function"), (fn2, "Second function")]  
for (f, arg) in listOfFuncs : f(arg)
```

You can return a function object from another function.

```
def FuncLair(produce) :  
    def fn1() : print('fn1 called')  
    def fn2() : print('fn2 called')  
    def fn3() : print('fn3 called')  
    if produce == 1 : return fn1  
    elif produce == 2 : return fn2  
    else : return fn3  
f = FuncLair(2) ; f()
```

## Function Attributes

- Python functions also have attributes.
- You can list them via the `dir()` built-in function.
- The attributes can be system-defined.
- Some of them can be user-defined as well.
- The `dir()` function also lists the user-defined attributes.

```
def testFunc():  
    print("I'm just a test function.")  
    testFunc.attr1 = "Hello"  
    testFunc.attr2 = 5  
testFunc()
```

- You can utilize the function attributes to archive state information instead of using any of the globals or nonlocals names.
- Unlike the nonlocals, attributes are accessible anywhere the function itself is, even from outside its code.



## Dictionary

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

- Dictionaries are optimized to retrieve values when the key is known.

### Create a dictionary

- Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.
- An item has a key and the corresponding value expressed as a pair, key: value.
- While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1: 'apple', 2: 'ball'})

# from sequence having each item as a pair
my_dict = dict([(1, 'apple'), (2, 'ball')])
```

---

We can also create a dictionary using the built-in function dict().

## Access Items in a Dictionary

- While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the get() method.
- The difference while using get() is that it returns None instead of KeyError, if the key is not found.

```
my_dict = {'name': 'Alice', 'age': 26}
# Output: Jack
print(my_dict['name'])
# Output: 26
print(my_dict.get('age'))
# Trying to access keys which doesn't exist throws error
# my_dict.get('address')
# my_dict['address']

#output
Alice
26
```

## Change or Add elements in a dictionary

- Dictionaries are mutable. We can add new items or change the value of existing items using assignment operator.
- If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```

my_dict = {'name': 'Alice', 'age': 26}
# update value
my_dict['age'] = 27
#Output: {'age': 27, 'name': 'Alice','}
print(my_dict)
# add item
my_dict['address'] = 'Downtown'
# Output: {'address': 'Downtown', 'age': 27, 'name': 'Alice','}
print(my_dict)

#Output
{'name': 'Alice',', 'age': 27}
{'name': 'Alice',', 'age': 27, 'address': 'Downtown'}

```

## Delete or remove elements from a dictionary

- We can remove a particular item in a dictionary by using the method pop(). This method removes an item with the provided key and returns the value.
- The method, popitem() can be used to remove and return an arbitrary item (key, value) from the dictionary. All the items can be removed at once using the clear() method.
- We can also use the del keyword to remove individual items or the entire dictionary itself.

```

# create a dictionary
squares = {1:1, 2:4, 3:9, 4:16, 5:25}
# remove a particular item
# Output: 16
print(squares.pop(4))
# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)
# remove an arbitrary item
# Output: (1, 1)
print(squares.popitem())
# Output: {2: 4, 3: 9, 5: 25}
print(squares)
# delete a particular item
del squares[5]
# Output: {2: 4, 3: 9}
print(squares)
# remove all items
squares.clear()

#Output
16
{1: 1, 2: 4, 3: 9, 5: 25}
(1, 1)
{2: 4, 3: 9, 5: 25}
{2: 4, 3: 9}
{}

```

## Python Dictionary Methods

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

- `clear()` - Remove all items from the dictionary.
- `copy()` - Return a shallow copy of the dictionary.
- `fromkeys(seq[, v])` - Return a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
- `get(key[,d])` - Return the value of `key`. If `key` does not exist, return `d` (defaults to `None`).
- `items()` - Return a new view of the dictionary's items (`key, value`).
- `keys()` - Return a new view of the dictionary's keys.
- `pop(key[,d])` - Remove the item with `key` and return its value or `d` if `key` is not found. If `d` is not provided and `key` is not found, raises `KeyError`.
- `popitem()` - Remove and return an arbitrary item (`key, value`). Raises `KeyError` if the dictionary is empty.
- `setdefault(key[,d])` - If `key` is in the dictionary, return its value. If not, insert `key` with a value of `d` and return `d` (defaults to `None`).
- `update([other])` - Update the dictionary with the `key/value` pairs from `other`, overwriting existing keys.
- `values()` - Return a new view of the dictionary's values

```
marks = {}.fromkeys(['Math', 'English', 'Science'], 0)
# Output: {'English': 0, 'Math': 0, 'Science': 0}
print(marks)
for item in marks.items():
    print(item)
# Output: ['English', 'Math', 'Science']
list(sorted(marks.keys()))
```

## Dictionary Comprehension

- Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python.
- Dictionary comprehension consists of an expression pair (key: value) followed by a for statement inside curly braces {}.
- Here is an example to make a dictionary with each item being a pair of a number and its square.

```
squares = {x: x*x for x in range(6)}  
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}  
print(squares)
```

#This code is equivalent to

```
squares = {}  
for x in range(6):  
    squares[x] = x*x
```

- A dictionary comprehension can optionally contain more for or if statements.
- An optional if statement can filter out items to form the new dictionary.
- Here are some examples to make dictionary with only odd items.

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}  
# Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}  
print(odd_squares)
```

## Other Dictionary Operations

**Dictionary Membership Test** - We can test if a key is in a dictionary or not using the keyword `in`. Notice that membership test is for keys only, not for values.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
# Output: True
print(1 in squares)
# Output: True
print(2 not in squares)
# membership tests for key only not value
# Output: False
print(49 in squares)
```

**Iterating Through a Dictionary** - Using a for loop we can iterate through each key in dictionary.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

## Built-in Functions with Dictionary

Built-in functions like `all()`, `any()`, `len()`, `cmp()`, `sorted()` etc. are commonly used with dictionary to perform different tasks.

- `all()` - Return True if all keys of the dictionary are true (or if the dictionary is empty).
- `any()` - Return True if any key of the dictionary is true. If the dictionary is empty, return False.
- `len()` - Return the length (the number of items) in the dictionary.
- `cmp()` - Compares items of two dictionaries.
- `sorted()` - Return a new sorted list of keys in the dictionary.

Here are some examples that use built-in functions to work with dictionary.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
# Output: 5
print(len(squares))
# Output: [1, 3, 5, 7, 9]
print(sorted(squares))
```

# Functions

## Introduction / Overview

- In Python, you have a couple of ways to make functions:
  - a) Use Def keyword: It creates a function object and assigns it to a name.
  - b) Use lambda: It creates an inline function and returns it as a result.
- A lambda function is a lightweight anonymous function. It can accept any number of arguments but can only have a single expression.

## What is lambda in Python?

- Lambda is an unnamed function. It provides an expression form that generates function objects.
- This expression form creates a function and returns its object for calling it later.

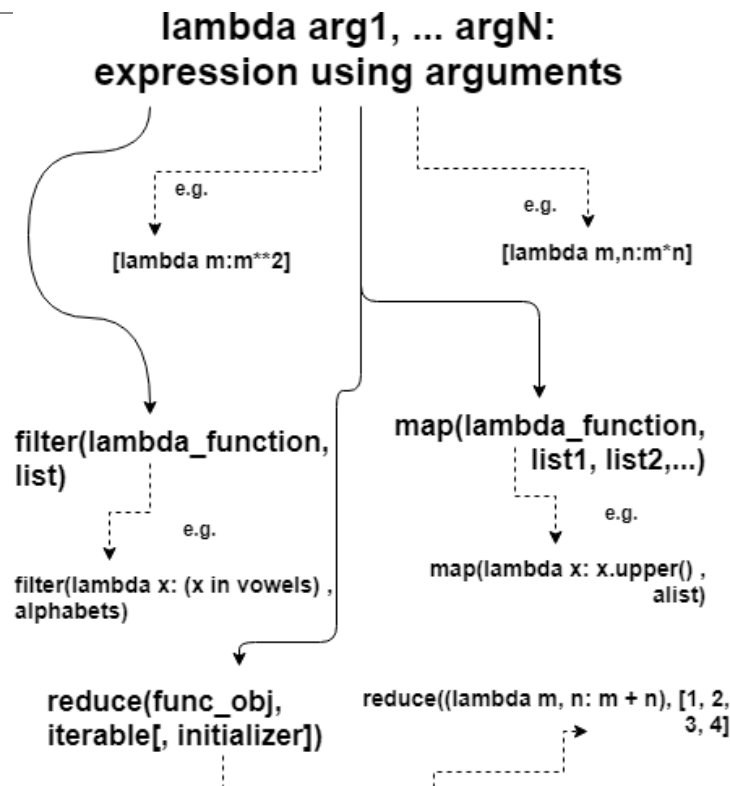


Image 32: Python – Lambda function

source: <https://images.app.goo.gl/6TPWQSmpJ1y1sNt26>

## How to create a lambda function?

### Syntax :

It has the following signature:

```
lambda arg1, arg2, ... argN: expression using arguments
```

- The body of a lambda function is akin to what you put in a def body's return statement. The difference here is that the result is a typed-expression, instead of explicitly returning it.



- Note: Lambda function can't include any statements. It only returns a function object which you can assign to any variable.
- The lambda statement can appear in places where the def is not allowed. For example –inside a list literal or a function call's arguments, etc.

### Example

#### lambda inside a list

```
list = [lambda m:m**2, lambda m,n:m*n, lambda m:m**4]
print(alist[0](10), alist[1](2, 20), alist[2](3))
# Output: 100 40 81
```

#### lambda inside a dictionary

```
key = 'm'
aDict = {'m': lambda x:2*x, 'n': lambda x:3*x}
print(aDict[key](9))
# Output: 18
```

## Why Use Lambda Functions in Python?

- A lambda function is not an absolute necessity in Python, but using a lambda function in certain situations definitely makes it a bit easier to write the code. Not just that, it also makes the written code a bit cleaner. Now, in what all situations using a lambda function is beneficial? Following are some of the situations where using a lambda function is preferred.
- Lambda functions in Python are very useful in defining the in-line functions where the regular functions, defined using the def keyword, won't work syntactically. Since a lambda function is an expression rather than a statement, it can be used in places

---

where a regular function is not allowed by the Python syntax, for instance, in places such as inside a Python list or in a function's call argument.

- As observed in the above example, the same operation performed by a regular function with the function body of at least three to four lines can be performed by a lambda function which only takes one line. Then, why use a long function to perform a simple operation when it can be done in a single line expression?
- So, to summarize, a lambda function behaves like a regular function, takes an argument, and returns a value but is not bound to any name or identifier. There is no need to use the return statement in a lambda function in Python; it will always return the value obtained by evaluating the lambda expression in Python.

### **Properties of Python Lambda Functions**

- Anonymous functions created using the lambda keyword can have any number of arguments, but they are syntactically restricted to just one expression, that is, they can have only one expression.
- Lambda function in Python can be used wherever a function object is required.
- Lambda functions do not require any return statement; they always return a value obtained by evaluating the lambda expression in Python.
- Python Lambda functions are widely used with some Python built-in functions such as map(), reduce(), etc. Extending Python lambda functions

### **Built in Functions**

- We can extend the utility of lambda functions by using it with the filter and map functions.
- It is possible by passing the lambda expression as an argument to another function. We refer to these methods as higher-order functions as they accept function objects as arguments.
- Python provides two built-in functions like filter(), map() which can receive lambda functions as arguments.

## Map functions over iterables – map()

- The map() function lets us call a function on a collection or group of iterables.
- We can also specify a Python lambda function in the map call as the function object.
- The map() function has the following

```
signature.      map(function_object,  
                    iterable1, iterable2,...)
```

- It expects variable-length arguments: first is the lambda function object, and rest are the iterables such as a list, dictionary, etc.

### What does the map() function do?

- The map function iterates all the lists (or dictionaries etc.) and calls the lambda function for each of their element.
- What does the map() function return?
- The output of map() is a list which contains the result returned by the lambda function for each item it gets called.
- Below is a simple example illustrating the use of map() function to convert elements of lists into uppercase.

```
# Python lambda demo to use map() for adding elements of two lists
```

```
alist = ['learn', 'python', 'step', 'by', 'step']  
output = list(map(lambda x: x.upper(), alist))
```

```
# Output: ['LEARN', 'PYTHON', 'STEP', 'BY', 'STEP']
```

```
print(output)
```

Let's have another example illustrating the use of `map()` function for adding elements of twolists.

```
# Python lambda demo to use map() for adding elements of two lists

list1 = [1, 2, 3, 4]
list2 = [100, 200, 300, 400]
output = list(map(lambda x, y: x+y , list1, list2))

# Output: [101, 202, 303, 404]

print(output)
```

### Select items in iterables – `filter()`

- The `filter()` function selects an iterable's (a list, dictionary, etc.) items based on a testfunction.
- We can also filter a list by using the Python lambda function as the function object.
- The `filter` function has the following signature.`filter(function_object, list)`
- It expects two parameters: first is the lambda function object and the second is a list.

### What does the `filter()` function do?

- The `filter` function iterates the list and calls the lambda function for each element.

### What does the `filter()` function return?

- It returns a final list containing items for which the lambda function evaluates to `True`.
- Below is a simple example illustrating the use of `filter()` function to determine vowels from the list of alphabets.

```
# Python lambda demo to filter out vowels from a list

alphabets = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
vowels = ['a', 'e', 'i', 'o', 'u']
output = list(filter(lambda x: (x in vowels) , alphabets))

# Output: ['a', 'e', 'i']
print(output)
```

### Aggregate items in iterables – reduce()

- The reduce method continuously applies a function on an iterable (such as a list) until there are no items left in the list. It produces a non-iterable result, i.e., returns a single value.
- This method helps in aggregating data from a list and returning the result. It can let us do a rolling calculation over successive pairs of values in a sequence.
- We can also pass a Python lambda function as an argument to the reduce method.
- The reduce() function has the following syntax.

```
reduce(func_obj, iterable[, initializer])
```

Below is a simple example where the reduce() method is computing the sum of elements in a list.

```
from functools import reduce
def fn(m, n) : return m + n
print(reduce((lambda m, n: m + n), [1, 2, 3, 4]))
print(reduce(fn, [1, 2, 3, 4]))

#After executing the above code, you see the following output.

10
10
```

# Modules

## What are modules in Python?

- Modules refer to a file containing Python statements and definitions.
- A file containing Python code, for e.g.: example.py, is called a module and its modulename would be example.
- We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.
- Generally, it is a good practice to create modules which have a fixed purpose. It increases readability and increases productivity and bug reporting.

Let us create a module. Type the following and save it as example.py.

```
# Python Module example
def add(a, b):
    """This program adds two numbers
    and return the result"""
    result = a + b
    return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

## Python Module: Mechanism

- For systems where Python is pre-installed or when installed using the system package manager such as apt-get, dnf, zypper, etc. or using package environment managers like Anaconda the following applies.
- When we import modules, the python interpreter locates them from three locations:
- The directory from the program is getting executed
- The directory specified in the PYTHONPATH variable (A shell variable or an environment variable)
- The default directory (It depends on the OS distribution.)
- The Flowchart for the above mechanism is as follows:

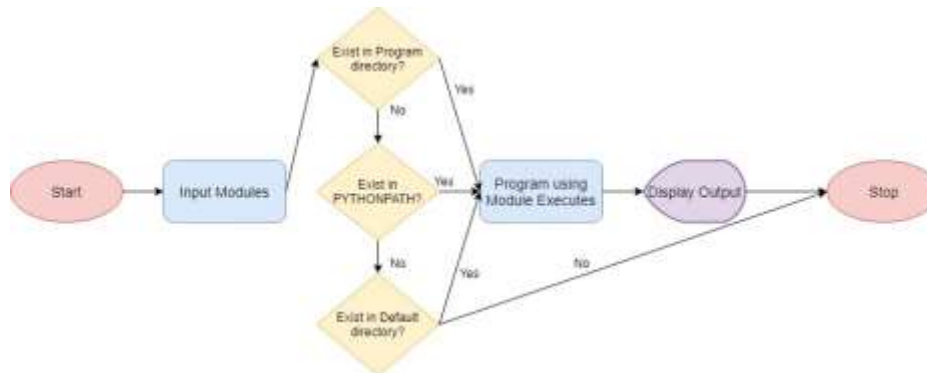


Image 33: Python – Module

source: <https://hyoseok-personality.tistory.com/entry/Week-1-10-Python-Module-Project>

## import modules in Python?

- We can import the definitions inside a module to another module or the interactive interpreter in Python.
- We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.

```
import example
```

- This does not enter the names of the functions defined in example directly in the current symbol table. It only enters the module name example there.
- Using the module name we can access the function using the dot operator.
- For example:

```
example.add(4, 5.5)
```

```
#output  
9.5
```

- Python has a ton of standard modules available.
- You can check out the full list of Python standard modules and what they are for. These files are in the Lib directory inside the location where you installed Python.
- Standard modules can be imported the same way as we import our user-defined modules.

**There are various ways to import modules. They are listed as follows.**

**import statement** - We can import a module using import statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example  
# to import standard module math  
import math  
print("The value of pi is", math.pi)
```



**Import with renaming** - We can import a module by renaming it as follows.

```
# import module by renaming it
import math as m
print("The value of pi is", m.pi)
```

We have renamed the math module as m. This can save us typing time in some cases.

- Note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

**Python from...import statement** - We can import specific names from a module without importing the module as a whole. Here is an example.

```
# import only pi from math module
from math import pi
print("The value of pi is", pi)
```

- We imported only the attribute pi from the module.
- In such case we don't use the dot operator. We could have imported multiple attributes as follows.

```
from math import pi, e
pi #3.141592653589793
e #2.718281828459045
```

**Import all names** - We can import all names(definitions) from a module using the following construct.

```
# import all names from the standard module math
from math import *
print("The value of pi is", pi)
```

We imported all the definitions from the math module. This makes all names except those beginning with an underscore, visible in our scope.

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

### Python Module Search Path

- While importing a module, Python looks at several places. Interpreter first looks for a built-in module then (if not found) into a list of directories defined in `sys.path`. The search is in this order.
- The current directory.
- `PYTHONPATH` (an environment variable with a list of directory).
- The installation-dependent default directory.

```
import sys
sys.path
```

Output:

```
['',
 'C:\\Python33\\Lib\\idlelib',
 'C:\\Windows\\system32\\python33.zip',
 'C:\\Python33\\DLLs',
 'C:\\Python33\\lib',
 'C:\\Python33',
 'C:\\Python33\\lib\\site-packages']
```

We can add modify this list to add our own path.

## Reloading a module

- The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.
- Suppose we have the following code in a module named my\_module.

```
# This module shows the effect of  
# multiple imports and reload  
print("This code got executed")
```

Now we see the effect of multiple imports.

```
import my_module
```

This code got executed

```
import my_module  
import my_module
```

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it. One way to do this is to restart the interpreter. But this does not help much.

- Python provides a neat way of doing this. We can use the `reload()` function inside the `importlib` to reload a module. This is how it's done.

```
import importlib  
importlib.reload(my_module)
```

This code got executed

```
import my_module  
imp.reload(my_module)
```

This code got executed

```
<module 'my_module' from './my_module.py'>
```

### The dir() built-in function

- We can use the dir() function to find out names that are defined inside a module.
- For example, we have defined a function add() in the module example that we had in the beginning.

```
dir()
```

Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module.

For example,

```
data = dir()  
print(data)
```

**#Output:**

```
['__annotations__', '__builtins__',  
 '__doc__', '__file__', '__loader__',  
 '__name__', '__package__', '__spec__']
```

# Package

## What are packages?

- We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.
- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.
- Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.
- Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.

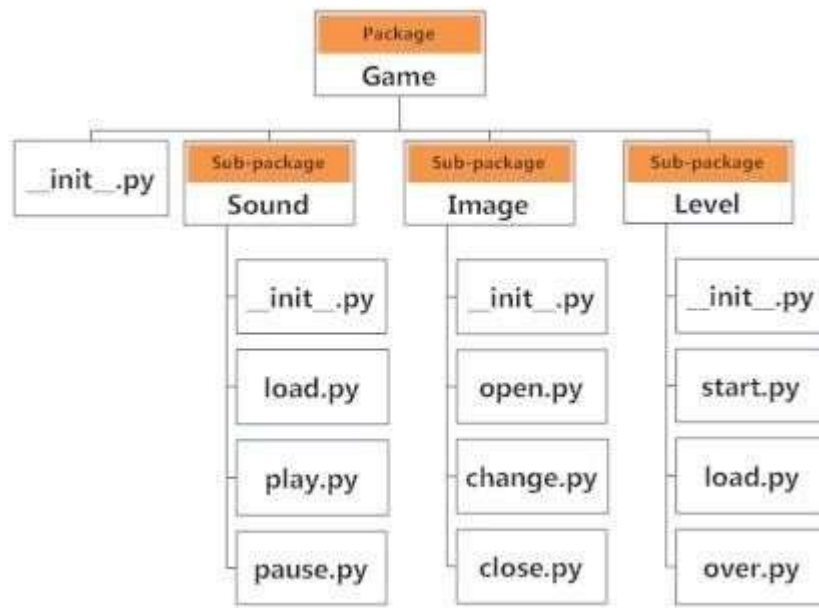


Image 34: Python – Packages

source: <https://images.app.goo.gl/2Txain7tnVquDbbi7>

## Importing module from a package

- We can import modules from packages using the dot (.) operator.
- For example, if want to import the start module in the above example, it is done asfollows.

```
import Game.Level.start
```

- Now if this module contains a function named `select_difficulty()`, we must use the fullname to reference it.

```
Game.Level.start.select_difficulty(2)
```

- If this construct seems lengthy, we can import the module without the package prefix as follows.

```
from Game.Level import start
```

- We can now call the function simply as follows.

```
start.select_difficulty(2)
```

- Yet another way of importing just the required function (or class or variable) from a module within a package would be as follows.

```
from Game.Level.start import select_difficulty
```

Now we can directly call this function.

```
select_difficulty(2)
```

Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding.

While importing packages, Python looks in the list of directories defined in `sys.path`, similar as for module search path.

# Input /Output

## Python Input, Output and Import

- Python provides numerous built-in functions that are readily available to us at the Pythonprompt.
- Some of the functions like input() and print() are widely used for standard input andoutput operations respectively. Let us see the output section first.
- Python Output Using print() function
- We use the print() function to output data to the standard output device (screen).
- An example of its use is given below.

```
print('This sentence is output to the screen')
```

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. Itdefaults 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).



## Output formatting

- Sometimes we would like to format our output to make it look attractive. This can be done by using the `str.format()` method. This method is visible to any string object.

```
x = 5; y = 10
print('The value of x is {} and y is {}'.format(x,y))
#Output
The value of x is 5 and y is 10
```

Here, the curly braces `{}` are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('bread', 'butter'))
print('I love {1} and {0}'.format('bread', 'butter'))
#Output
I love bread and butter
I love butter and bread
```

We can even use keyword arguments to format the string.

```
print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))
Hello John, Goodmorning
```

We can also format strings like the old `sprintf()` style used in C programming language. We use the `%` operator to accomplish this.

```
x = 12.3456789
print('The value of x is %3.2f' %x)
#The value of x is 12.35
print('The value of x is %3.4f' %x)
#The value of x is 12.3457
```

## Python Input

- The value of variables was 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 `eval` takes it further. It can evaluate even expressions, provided the input is a string

```
int('2+3')
eval('2+3')
```

## What is a file?

- File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since, random access memory (RAM) is volatile which loses its data when

---

computer is turned off, we use files for future use of the data.

- When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.
- In Python, file processing takes place in the following order.
  - Open a file that returns a filehandle.
  - Use the handle to perform read or write action.
  - Close the filehandle.
- Before you do a read or write operation to a file in Python, you need to open it first. And as the read/write transaction completes, you should close it to free the resources tied with the file.

## How to open a file?

- Python has a built-in function `open()` to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
# open file in current directory
f = open("test.txt")

# specifying full path
f = open("C:/Python33/README.txt")
```

- We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.
- The default is reading in text mode. In this mode, we get strings when reading from the file.
- On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

- Python open() file method

```
file object = open(file_name [, access_mode][, buffering])
```

Below are the parameter details.

- <access\_mode>- It's an integer representing the file opening mode, e.g., read, write, append, etc. It's an optional parameter. By default, it is set to read-only <r>. In this mode, we get data in text form after reading from the file.
- On the other hand, the binary mode returns bytes. It's preferable for accessing the non-text files like an image or the Exe files. See the table in the next section. It lists down the available access modes.
- <buffering>- The default value is 0, which means buffering won't happen. If the value is 1, then line buffering will take place while accessing the file. If it's higher than 1, then the buffering action will run as per the buffer size. In the case of a negative value, the default behavior is considered.
- <file\_name>- It's a string representing the name of the file you want to access.

### Python File Modes

MODES	DESCRIPTION
<r>	It opens a file in read-only mode while the file offset stays at the root.
<rb>	It opens a file in (binary + read-only) modes. And the offset remains at the root level.
<r+>	It opens the file in both (read + write) modes while the file offset is again at the root level.

<rb+>	It opens the file in (read + write + binary) modes. The file offset is again at the rootlevel.
<w>	It allows write-level access to a file. If the file already exists, then it'll get overwritten. It'll create a new file if the same doesn't exist.
<wb>	Use it to open a file for writing in binary format. Same behavior as for write-onlymode.
<w+>	It opens a file in both (read + write) modes. Same behavior as for write-only mode.
<wb+>	It opens a file in (read + write + binary) modes. Same behavior as for write-onlymode.
<a>	It opens the file in append mode. The offset goes to the end of the file. If the file doesn't exist, then it gets created.
<ab>	It opens a file in (append + binary) modes. Same behavior as for append mode.
<a+>	It opens a file in (append + read) modes. Same behavior as for append mode.
<ab+>	It opens a file in (append + read + binary) modes. Same behavior as for appendmode.

```

# equivalent to 'r' or 'rt'
f = open("test.txt")

# write in text mode
f = open("test.txt", 'w')

# read and write in binary mode
f = open("img.bmp", 'r+b')

```

- Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).
- Moreover, the default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux.
- So, we must not also rely on the default encoding or else our code will behave differently in different platforms.
- Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt", mode = 'r', encoding = 'utf-8')
```

### The Python file object attributes

- When you call the Python `open()` function, it returns an object, which is the filehandle. Also, you should know that Python files have several linked attributes. And we can make use of the filehandle to list the attributes of a file.
- For more information on file attributes, please run down through the below table.

Attribute	Description
<file.closed>	For a closed file, it returns true whereas false otherwise.
<file.mode>	It returns the access mode used to open a file.
<file.name>	Returns the name of a file

<code>&lt;file.softspace&gt;</code>	It returns a boolean to suggest if a space char will get added before printing another value in the output of a <code>&lt;print&gt;</code> command.
-------------------------------------	---

Example: Python file attribute in action

```
#Open a file in write and binary mode.
fob = open("app.log", "wb")

#Display file name.
print("File name: ", fob.name)

#Display state of the file.
print("File state: ", fob.closed)

#Print the opening mode.
print("Opening mode: ", fob.mode)

#Output the softspace value.
print("Softspace flag: ", fob.softspace)
```

## How to close a file?

- It's always the best practice to close a file when your work gets finished. However, Python runs a garbage collector to clean up the unused objects. But you must do it on your own instead leave it for the GC.
- The `close()` file method
- Python provides the `<close()>` method to close a file.

- While closing a file, the system frees up all resources allocated to it. And it's rather easy to achieve.
- Close operation in Python
- The most basic way is to call the Python `close()` method.

```
f = open("app.log", encoding = 'utf-8')
# do file operations.
f.close()
```

### Close with try-catch

Say, if an exception occurs while performing some operations on the file. In such a case, the code exits without closing the file. So it's better to put the code inside a `<try-finally>` block.

```
try:
    f = open('app.log', encoding = 'utf-8')
    # do file operations.
finally:
    f.close()
```

So, even if there comes an exception, the above code will make sure your file gets appropriately closed.

### Auto close using 'with

Another way to close a file is by using the `WITH` clause. It ensures that the file gets closed when the block inside the `WITH` clause executes. The beauty of this method is that it doesn't require to call the `close()` method explicitly.

```
with open('app.log', encoding = 'utf-8') as f:
    #do any file operation.
```



---

## Perform Write operation

- While you get ready for writing data to a file, first of all, open it using a mode (read/write/append). View the list of all available file modes [here](#).
- You can even do the same using the append mode. Also, if you've used the <w> mode, then it'll erase the existing data from the file. So you must note this fact while you choose it.
- The write() file method
- Python provides the write() method to write a string or sequence of bytes to a file. This function returns a number, which is the size of data written in a single Write call.

### Example: Read/Write to a File in Python

```
with open('app.log', 'w', encoding = 'utf-8') as f:

    #first line
    f.write('my first file\n')

    #second line
    f.write('This file\n')

    #third line
    f.write('contains three lines\n')
with open('app.log', 'r', encoding = 'utf-8') as f:
    content = f.readlines()
    for line in content:
        print(line)
```

---

## Perform Read operation

- To read data from a file, first of all, you need to open it in reading mode. Then, you can call any of the methods that Python provides for reading from a file.
- Usually, we use Python <read(size)> function to read the content of a file up to the size. If you don't pass the size, then it'll read the whole file.
- Example: Read from a File in Python

```
with open('app.log', 'w', encoding = 'utf-8') as f:
    #first line
    f.write('my first file\n')
    #second line
    f.write('This file\n')
    #third line
    f.write('contains three lines\n')
    f = open('app.log', 'r', encoding = 'utf-8')
    # read the first 10 data
    print(f.read(10))
    #'my first f'
    print(f.read(4))
    #'ile\n'
    # read in the rest till end of file
    print(f.read())
    #'This file\ncontains three lines\n'
    # further reading returns empty string#''
    print(f.read())
```

## Set File offset in Python

- Tell() Method - This method gives you the current offset of the file pointer in a file.

```
file.tell()
#The tell() method doesn't require any argument
```

- Seek() Method - This method can help you change the position of a file pointer in a file.

```
file.seek(offset[, from])  
#The <offset> argument represents the size of the displacement.  
#The <from> argument indicates the start point
```

## Renaming and deleting files in Python

- While you were using the <read/write> functions, you may also need to <rename/delete> a file in Python. So, there comes a <os> module in Python, which brings the support of file <rename/delete> operations.
- So, to continue, first of all, you should import the <os> module in your Python script.
- The rename() file method

```
#Syntax  
os.rename(cur_file, new_file)
```

- The <rename()> method takes two arguments, the current filename and the new filename.
- Following is the example to rename an existing file <app.log> to <app1.log>.

```
import os  
#Rename a file from <app.log> to <app1.log>  
os.rename( "app.log", "app1.log" )  
The remove() file method  
os.remove(file_name)
```

The <remove()> method deletes a file which it receives in the argument. Following is the example to delete an existing file, the <app1.log>.Example:

```
import os  
#Delete a file <app1.log>  
os.remove( "app1.log" )
```

## Python File object methods

- So far, we've only shared with you a few of the functions that you can use for filehandling in Python. But there is more to the story of Python file handling.
- Python's `open()` method returns an object which we call as the filehandle. Python adds ano. of functions that we can call using this object.

Function	Description
<code>&lt;file.close()&gt;</code>	Close the file. You need to reopen it for further access.
<code>&lt;file.flush()&gt;</code>	Flush the internal buffer. It's same as the <code>&lt;stdio&gt;</code> 's <code>&lt;fflush()&gt;</code> function.
<code>&lt;file.fileno()&gt;</code>	Returns an integer file descriptor.
<code>&lt;file.isatty()&gt;</code>	It returns true if file has a <code>&lt;tty&gt;</code> attached to it.
<code>&lt;file.next()&gt;</code>	Returns the next line from the last offset.
<code>&lt;file.read(size)&gt;</code>	Reads the given no. of bytes. It may read less if EOF is hit.
<code>&lt;file.readline(size)&gt;</code>	It'll read an entire line (trailing with a new line char) from the file.
<code>&lt;file.readlines(size_hint)&gt;</code>	It calls the <code>&lt;readline()&gt;</code> to read until EOF. It returns a list of lines read from the file. If you pass <code>&lt;size_hint&gt;</code> , then it reads lines equalling the <code>&lt;size_hint&gt;</code> bytes.
<code>&lt;file.seek(offset [,from])&gt;</code>	Sets the file's current position.
<code>&lt;file.tell()&gt;</code>	Returns the file's current position.

<code>&lt;file.truncate(size)&gt;</code>	Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.
<code>&lt;file.write(string)&gt;</code>	It writes a string to the file. And it doesn't return any value.
<code>&lt;file.writelines(sequence)&gt;</code>	Writes a sequence of strings to the file. The sequence is possibly an iterable object producing strings, typically a list of strings.

## Python Copy File – 9 Ways

- `shutil copyfile()` method
- `shutil copy()` method
- `shutil copyfileobj()` method
- `shutil copy2()` method
- `os popen` method
- `os system()` method
- `threading Thread()` method
- `subprocess call()` method
- `subprocess check_output()` method

### 1. `shutil copyfile()` method

- This method copies the content of the source to the destination only if the target is writable. If you don't have the right permissions, then it will raise an `IOError`.

- 
- It works by opening the input file for reading while ignoring its file type.
  - Next, it doesn't treat special files any differently and won't create their clones.
  - The `copyfile()` method makes use of lower-level function `copyfileobj()` underneath. It takes file names as arguments, opens them and passes file handles to `copyfileobj()`. There is one optional third argument in this method which you can use to specify the buffer length. It'll then open the file for reading in chunks of the specified buffer size. However, the default behavior is to read the entire file in one go.

```
copyfile(source_file, destination_file)
```

- Following are the points to know about the `copyfile()` method.
- It copies the contents of the source to a file named as the destination.
- If the destination isn't writable, then the copy operation would result in an

`IOError` exception.

- It will return the `SameFileError` if both the source and destination files are the same.
- However, if the destination pre-exists with a different name, then the copy will overwrite its content.
- Error 13 will occur if the destination is a directory which means this method won't copy to a folder.
- It doesn't support copying files such as character or block devices and the pipes.

```
# Python Copy File - Sample Code
from shutil import copyfile
from sys import exit
source = input("Enter source file with full path: ")
target = input("Enter target file with full path: ")
# adding exception handling
try:
    copyfile(source, target)
except IOError as e:
    print("Unable to copy file. %s" % e)
    exit(1)
except:
    print("Unexpected error:", sys.exc_info())
    exit(1)
print("\nFile copy done!\n")
while True:
    print("Do you like to print the file ? (y/n): ")
    check = input()
    if check == 'n':
        break
    elif check == 'y':
        file = open(target, "r")
        print("\nHere follows the file content:\n")
        print(file.read())
        file.close()
        print()
        break
    else:
        continue
```

## 2. shutil copy() method

```
copyfile(source_file, [destination_file or dest_dir])
```

- The copy() method functions like the “cp” command in Unix. It means if the target is a folder, then it'll create a new file inside it with the same name (basename) as the source file. Also, this method will sync the permissions of the target file with the source after copying its content. It too throws the SameFileError if you are copying the same file.

```
import os
import shutil

source = 'current/test/test.py'
target = '/prod/new'

assert not os.path.isabs(source)
target = os.path.join(target, os.path.dirname(source))

# create the folders if not already exists
os.makedirs(target)

# adding exception handling
try:
    shutil.copy(source, target)
except IOError as e:
    print("Unable to copy file. %s" % e)
except:
    print("Unexpected error:", sys.exc_info())
```

### copy() vs copyfile()

- The copy() also sets the permission bits while copying the contents whereas the copyfile() only copies the data.
- The copy() will copy a file if the destination is a directory whereas the copyfile() will fail with error 13.
- Interestingly, the copyfile() method utilizes the copyfileobj() method in its implementation whereas the copy() method makes use of the copyfile() and copymode() functions in turn.
- Point-3 makes it apparent that copyfile() would be a bit faster than the copy() as the latter has an additional task (preserving the permissions) at hand.



---

### 3. shutil copyfileobj() method

- This method copies the file to a target path or file object. If the target is a file object, then you need to close it explicitly after the calling the copyfileobj(). It assumes an optional argument (the buffer size) which you can use to supply the buffer length. It is the number of bytes kept in memory during the copy process. The default size that system use is 16KB.

```
from shutil import copyfileobj
status = False
if isinstance(target, string_types):
    target = open(target, 'wb')
    status = True
try:
    copyfileobj(self.stream, target, buffer_size)
finally:
    if status:
        target.close()
```

### 4. shutil copy2() method

- However, the copy2() method functions like the copy(). But it also gets the access and modification times added in the meta-data while copying the data. Copying the same file would result in SameFileError.

```
from shutil import *
import os
import time
from os.path import basename

def displayFileStats(filename):
    file_stats = os.stat(basename(filename))
    print('\tMode      : ', file_stats.st_mode)
    print('\tCreated   : ', time.ctime(file_stats.st_ctime))
    print('\tAccessed  : ', time.ctime(file_stats.st_atime))
    print('\tModified  : ', time.ctime(file_stats.st_mtime))

os.mkdir('test')

print('SOURCE:')
displayFileStats(__file__)

copy2(__file__, 'testfile')

print('TARGET:')
displayFileStats(os.path.realpath(os.getcwd() + './test/testfile'))
```

## copy() vs copy2()

- The copy() only sets permission bits whereas copy2() also updates the file metadata with timestamps.
- The copy() method calls copyfile() and copymode() internally whereas copy2() replaces the call to copymode() with copystat().

## 5. shutil.copymode()

```
copymode(source, target, *, follow_symlinks=True)
```

- It intends to copy the permission bits from source to the target files.
- The file contents, owner, and group remain unchanged. The arguments passed are strings.
- If the follow\_symlinks arg is false and the first two args are symlinks, then copymode() will try to update the target link, not the actual file it is pointing.

## 6. shutil.copystat()

```
copystat(source, target, *, follow_symlinks=True)
```

- It attempts to preserve the permission bits, last used time/update time, and flags of the target file.
- The copystat() includes the extended attributes while copying on Linux. The file contents/owner/group remain unchanged.
- If the follow\_symlinks arg is false and the first two args are symlinks, then copystat() will update them, not the files they point.

## 7. os.popen() method

- This method creates a pipe to or from the command. It returns an open file object which connects to a pipe. You can use it for reading or writing according to the file mode, i.e., 'r' (default) or 'w'.

```
os.popen(command[, mode[, bufsize]])  
mode - It can be 'r' (default) or 'w'.
```

- bufsize – If its value is 0, then no buffering will occur. If the bufsize is 1, then line buffering will take place while accessing the file. If you provide a value greater than 1, then buffering will occur with the specified buffer size. However, for a negative value, the system will assume the default buffer size.

```
#For Windows OS.  
import os  
os.popen('copy 1.txt.py 2.txt.py')  
  
#For Linux OS.  
import os  
os.popen('cp 1.txt.py 2.txt.py')
```

## 8. os system() method

- The system() method allows you to instantly execute any OS command or a script in the subshell.
- You need to pass the command or the script as an argument to the system() call. Internally, this method calls the standard C library function.
- Its return value is the exit status of the command.

```
#For Windows OS.  
import os  
os.system('copy 1.txt.py 2.txt.py')  
  
#For Linux OS.  
import os  
os.system('cp 1.txt.py 2.txt.py')
```

## Python file copy using threading library in Async manner

- If you want to copy a file asynchronously, then use the below method. In this, we've used Python's threading module to run the copy operation in the background.
- While using this method, please make sure to employ locking to avoid deadlocks. You may face it if your application is using multiple threads reading/writing a file.

```
import shutil  
from threading import Thread  
  
src="1.txt.py"  
dst="3.txt.py"  
  
Thread(target=shutil.copy, args=[src, dst]).start()
```

## 9. Use subprocess's call() method to copy a file in Python

- The subprocess module gives a simple interface to work with child processes. It enables us to launch subprocesses, attach to their input/output/error pipes, and retrieve the return values.
- The subprocess module aims to replace the legacy modules and functions like – os.system, os.spawn\*, os.popen\*, popen2.\*.
- It exposes a call() method to invoke system commands to execute user tasks.

```
import subprocess
src="1.txt.py"
dst="2.txt.py"
cmd='copy "%s" "%s"' % (src, dst)
status = subprocess.call(cmd, shell=True)
if status != 0:
    if status < 0:
        print("Killed by signal", status)
    else:
        print("Command failed with return code - ", status)
else:
    print('Execution of %s passed!\n' % cmd)
```

## 10. Use subprocess's check\_output() method to copy a file in Python

- With subprocess's check\_output() method, you can run an external command or a program and capture its output. It also supports pipes.

```
import os, subprocess
src=os.path.realpath(os.getcwd() + "./1.txt.py")
dst=os.path.realpath(os.getcwd() + "./2.txt.py")
cmd='copy "%s" "%s"' % (src, dst)
status = subprocess.check_output(['copy', src, dst], shell=True)
print("status: ", status.decode('utf-8'))
```

# Exception Handling

## Exception Handling: Error vs. Exception

### What is Error?

- The error is something that goes wrong in the program, e.g., like a syntactical error.
- It occurs at compile time. Let's see an example.

```
if a<5
File "<interactive input>", line 1
    if a < 5
        ^
SyntaxError: invalid syntax
```

### What is Exception?

- The errors also occur at runtime, and we know them as exceptions. An exception is an event which occurs during the execution of a program and disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters an error situation that it can't cope with, it raises an exception.
- When a Python script raises an exception, it creates an exception object.
  - Usually, the script handles the exception immediately. If it doesn't do so, then the program will terminate and print a traceback to the error along with its whereabouts.

```
1 / 0
Traceback (most recent call last):
File "<string>", line 301, in run code
File "<interactive input>", line 1, in <module>
ZeroDivisionError: division by zip
```

## Handle Exceptions with Try-Except

### What is Try-Except Statement?

- We use the try-except statement to enable exception handling in Python programs.
- Inside the try block, you write the code which can raise an exception.
- And the code that handles or catches the exception, we place in the except clause.
- Python Exception Handling Syntax
- Following is the syntax of a Python try-except-else block.

```
try:
    You do your operations here;
    .....
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
    .....
else:
```

Here is a checklist for using the Python try statement effectively.

- If there is no exception then execute this block.
- A single try statement can have multiple except statements depending on the requirement. In this case, a try block contains statements that can throw different types of exceptions.

- We can also add a generic except clause which can handle all possible types of exceptions.
- We can even include an else clause after the except clause. The instructions in the elseblock will execute if the code in the try block doesn't raise an exception.
- We'll perform a WRITE operation on it. Upon execution, it'll throw an exception.

```
try:
    fob = open("test", "r")
    fob.write("It's my test file to verify exception handling in Python!!")
except IOError:
    print "Error: can't find the file or read data"
else:
    print "Write operation is performed successfully on the file"

#The above code produces the following output.

Error: can't find file or read data
```

## Handling All Types of Exceptions with Except

- If we use a bare “except” clause, then it would catch all types of exceptions.
- However, neither it's a good programming practice nor does anyone recommend it.
- It is because that such a Python try-except block can handle all types of exceptions. But it'll not help the programmer to find what exception caused the issue.
- You can go through the below code to see how to catch all exceptions.
- Example

```
try:
    You do your operations here;
    .....
except:
    If there is any exception, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```



## Handling Multiple Exceptions with Except

- We can define multiple exceptions with the same except clause. It means that if the Python interpreter finds a matching exception, then it'll execute the code written under the except clause.
- In short, when we define except clause in this way, we expect the same piece of code to throw different exceptions. Also, we want to take the same action in each case.
- Please refer to the below example.

- Example

```
try:
    You do your operations here;
    .....
except (Exception1[, Exception2[, ...ExceptionN]]):
    If there is any exception from the given exception list,
    then execute this block.
    .....
else:
    If there is no exception then execute this block
```

## Handle Exceptions with Try-Finally

### What is Try-Finally Statement?

- We can also enable Python exception handling with the help of try-finally statement.
- With try block, we also have the option to define the “finally” block. This clause allows defining statements that we want to execute, no matters whether the try block has raised an exception or not.
- This feature usually comes in the picture while releasing external resources.
- Here is the coding snippet for help.

```
try:
    You do your operations here;
    .....
Due to any exception, this may be skipped.
finally:
    This would always be executed.
    .....
```

## Raise Exception with Arguments

### What is Raise?

- We can forcefully raise an exception using the raise keyword.
- We can also optionally pass values to the exception and specify why it has occurred.
- Raise Syntax-

```
raise [Exception [, args [, traceback]]]
```

- Where, Under the “Exception” – specify its name.
- The “args” is optional and represents the value of the exception argument.
- The final argument, “traceback,” is also optional and if present, is the traceback object used for the exception.
- Let’s take an example to clarify this.
- Raise Example

```
raise MemoryError
Traceback (most recent call last):
...
MemoryError

>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument

>>> try:
    a = int(input("Enter a positive integer value: "))
    if a <= 0:
        raise ValueError("This is not a positive number!!")
except ValueError as ve:
    print(ve)
```

Following Output is displayed if we enter a negative number:

Enter a positive integer: -5

This is not a positive number!!

## Create Custom Exceptions

### What is a Custom Exception?

- A custom exception is one which the programmer creates himself.
- He does it by adding a new class. The trick here is to derive the custom exception class from the base exception class.
- Most of the built-in exceptions do also have a corresponding class.
- Create Exception Class in Python

```
class UserDefinedError(Exception):
...     pass

raise UserDefinedError
Traceback (most recent call last):
...
__main__.UserDefinedError

raise UserDefinedError("An error occurred")
Traceback (most recent call last):
...
__main__.UserDefinedError: An error occurred
```

## Python Built-in Exceptions

Exception	Cause of Error
AirthmeticError	For errors in numeric calculation.
AssertionError	If the assert statement fails.
AttributeError	When an attribute assignment or the reference fails.
EOFError	If there is no input or the file pointer is at EOF.
Exception	It is the base class for all exceptions.
EnvironmentError	For errors that occur outside the Python environment.
FloatingPointError	It occurs when the floating point operation fails.
GeneratorExit	If a generator's <close()> method gets called.

ImportError	It occurs when the imported module is not available.
IOError	If an input/output operation fails.
IndexError	When the index of a sequence is out of range.
KeyError	If the specified key is not available in the dictionary.
KeyboardInterrupt	When the user hits an interrupt key (Ctrl+c or delete).
MemoryError	If an operation runs out of memory.
NameError	When a variable is not available in local or global scope.
NotImplementedError	If an abstract method isn't available.
OSError	When a system operation fails.
OverflowError	It occurs if the result of an arithmetic operation exceeds the range.
ReferenceError	When a weak reference proxy accesses a garbage collected reference.
RuntimeError	If the generated error doesn't fall under any category.
StandardError	It is a base class for all built-in exceptions except <StopIteration> and <SystemExit>.
StopIteration	The <next()> function has no further item to be returned.

SyntaxError	For errors in Python syntax.
IndentationError	It occurs if the indentation is not proper.
TabError	For inconsistent tabs and spaces.
SystemError	When interpreter detects an internal error.
SystemExit	The <sys.exit()> function raises it.
TypeError	When a function is using an object of the incorrect type.
UnboundLocalError	If the code using an unassigned reference gets executed.
UnicodeError	For a Unicode encoding or decoding error.
ValueError	When a function receives invalid values.
ZeroDivisionError	If the second operand of division or modulo operation is zero.

---

# Object Oriented Python

- Object-oriented programming (OOP) is a way of organizing a programme by grouping similar characteristics and activities into separate objects. This lesson will teach you the fundamentals of Python object-oriented programming.
- Objects are similar to the components of a system in terms of concept. Consider a programme like a factory assembly line. A system component processes some material at each stage of the assembly line, eventually changing raw material into a finished product.
- Data, such as raw or preprocessed materials at each stage on an assembly line, and behavior, such as the action each assembly line component performs, are both contained in an object.

## Principles of object oriented programming

Object Oriented Programming (OOP) is a programming paradigm that emphasizes objects over actions and data over logic. To be object-oriented, a programming language must include a system for dealing with classes and objects, as well as the implementation and usage of object-oriented ideas and concepts such as inheritance, abstraction, encapsulation, and polymorphism.

## OOPs (Object-Oriented Programming System)

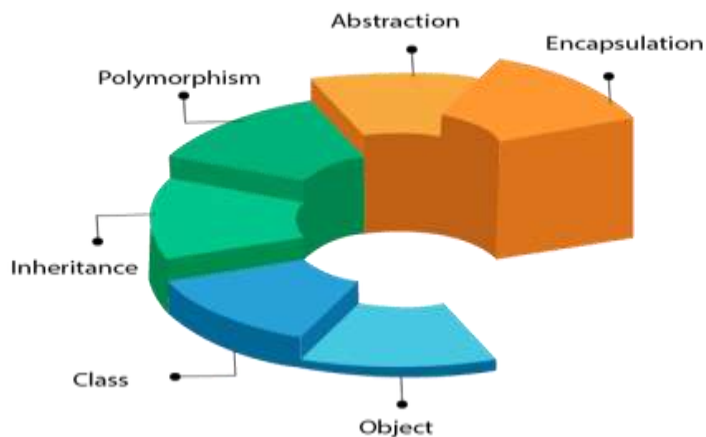


Image 35: Python – Object Oriented Programming

Source: <https://images.app.goo.gl/e46rr8hhRHHBwwMb8>

Let's take a quick look at each of the pillars of object-oriented programming:

### Encapsulation

This feature conceals unneeded information and simplifies programme structure management. The implementation and state of each object are buried behind well-defined boundaries, resulting in a clean and straightforward user interface. Making the data private is one approach to do this.

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).
- It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
- To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.



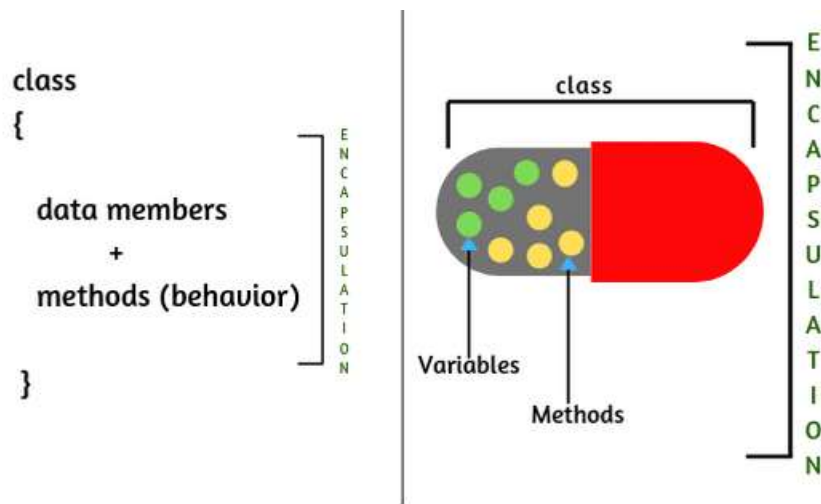


Image 36: Python – Encapsulation

source: <https://images.app.goo.gl/AV3Z5jfN6v47DTRH8>

## Inheritance

We can capture a hierarchical relationship between classes and objects using inheritance, also known as generalization. A 'fruit,' for example, is a generalization of 'orange.' From the standpoint of code reuse, inheritance is really valuable.

- Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more.
- Inheritance is a powerful feature in object oriented programming.
- It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

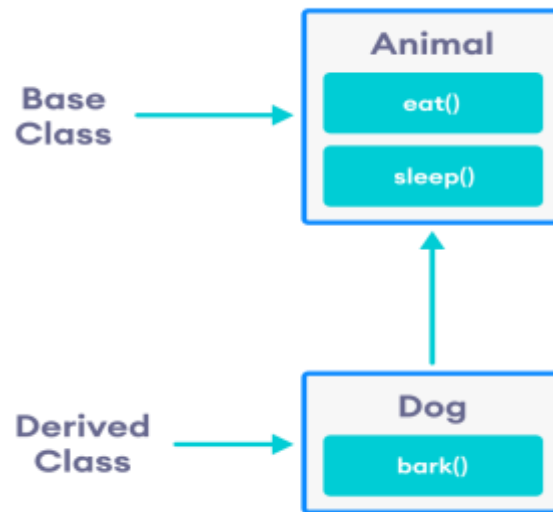


Image 37: Python – Inheritance

source: <https://images.app.goo.gl/oqWLAU9RdhstmkjR6>

## Abstraction

This attribute allows us to hide the intricacies of a concept or object and just show the main aspects. For example, a scooter driver is aware that pushing the horn produces sound, but he is unaware of how the sound is formed.

## Polymorphism

Many forms is what polymorphism means. That is, a thing or activity exists in several forms or modes. Constructor overloading in classes is a nice illustration of polymorphism.

## Class in Python

- Primitive data structures—like numbers, strings, and lists—are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively. What if you want to represent something more complex?

- For example, let's say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

```
Python
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

- There are a number of issues with this approach.
- First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where the `kirk` list is declared, will you remember that the element with index 0 is the employee's name?
- Second, it can introduce errors if not every employee has the same number of elements in the list. In the `mccoy` list above, the age is missing, so `mccoy[1]` will return "Chief Medical Officer" instead of Dr. McCoy's age.
- A great way to make this type of code more manageable and more maintainable is to use classes.

## Classes vs. Instances

- Classes are used to create user-defined data structures. Classes define functions called methods, which identify the behaviors and actions that an object created from the class can perform with its data.
- Now create a `Dog` class that stores some information about the characteristics and behaviors that an individual dog can have.
- A class is a blueprint for how something should be defined. It doesn't actually contain any data. The `Dog` class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.
- While the class is the blueprint, an instance is an object that is built from a class and contains real

data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

### How to define a class

All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.

Here's an example of a Dog class:

```
Python

class Dog:
    pass
```

The body of the Dog class consists of a single statement: the pass keyword. pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

- The Dog class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Dog objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed. To keep things simple, we'll just use name and age.
- The properties that all Dog objects must have are defined in a method called `__init__()`. Every time a new Dog object is created, `__init__()` sets the initial state of the object by assigning the values of the object's properties. That is, `__init__()` initializes each new instance of the class.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- In the body of `__init__()`, there are two statements using the `self` variable:
- Attributes created in `__init__()` are called instance attributes. An instance attribute's value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.
- On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `__init__()`.

In the body of `__init__()`, there are two statements using the `self` variable:

1. `self.name = name` creates an attribute called `name` and assigns to it the value of the `name` parameter.
2. `self.age = age` creates an attribute called `age` and assigns to it the value of the `age` parameter.

## Self Parameter

- Methods or functions should have `self` as first parameter.
- When objects are instantiated, the object itself is passed into the `self` parameter.
- The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named `self`, you can call it whatever you like, but it has to be the first parameter of any function in the class.

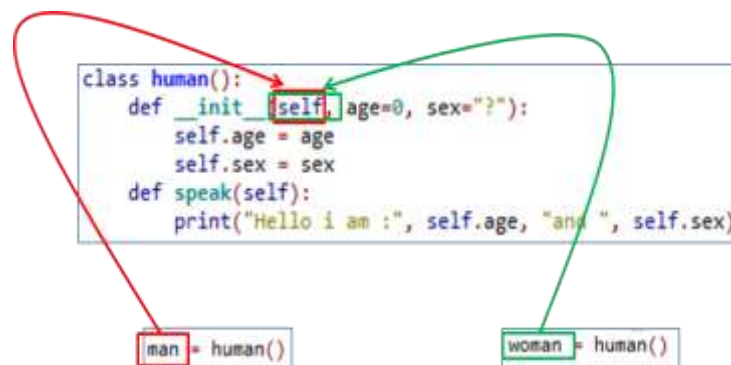


Image 38: Python – Self Parameter

source: <https://images.app.goo.gl/dGTBFVpAFDa8qf7H8>

## Constructors

- It is a special method that is automatically invoked right after a new object is created.
- It is used to initialize the attribute values of new object created.
- `__init__()` is a reserved method in python classes. It is called as a constructor in object oriented terminology.
- This method is called when an object is created from a class and it allows the class to initialize the attributes of the class

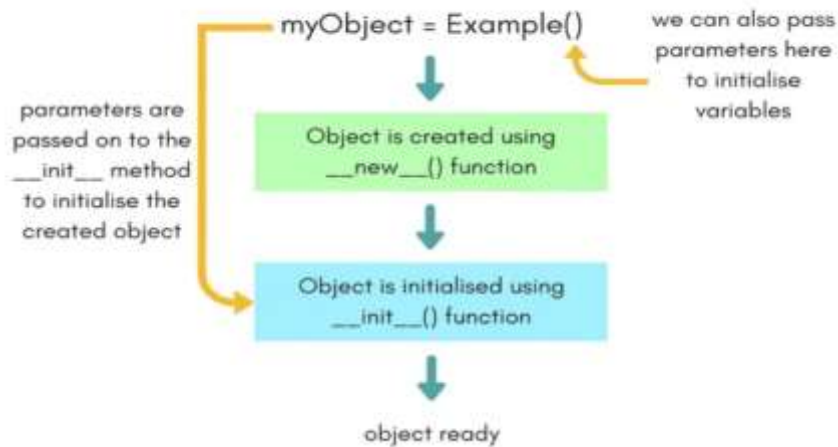


Image 39: Python – Illustration of Constructor

source: <https://www.studytonight.com/python/constructors-in-python>

## Syntax for constructor declaration

```
def __init__(self):  
    # body of constructor
```

## Different types of constructors

### Default Constructor

- Doesn't have any arguments
- It has only one argument which is a reference to the instance being constructed

### Parameterized Constructor

- constructor with parameters
- first argument is reference to instance

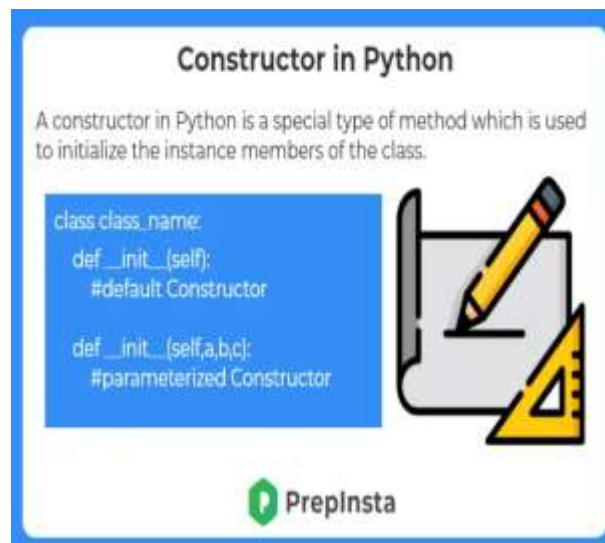


Image 40: Python – Constructor

source: <https://images.app.goo.gl/FWHxRVxD7qWEHy9R8>

## Class variables and Instance variables

**Class Variables** — Declared inside the class definition (but outside any of the instance methods). They are not tied to any particular object of the class, hence shared across all the objects of the class. Modifying a class variable affects all objects instance at the same time.

**Instance Variable** — Declared inside the constructor method of class (the `__init__` method). They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.

Example:

```
class Car:
    wheels = 4 # Class variable
    def __init__(self, name):
        self.name = name #Instance variable
```

### Destructors in Python

- When an object is destroyed, destructors are invoked. Destructors aren't as important in Python as they are in C++ because Python has a garbage collector that handles memory management for you.
- In Python, the `del ()` function is known as a destructor method. It is called after all references to the object have been destroyed i.e when an object is garbage collected.
- Syntax:

```
def __del__(self):
    # body of destructor
```



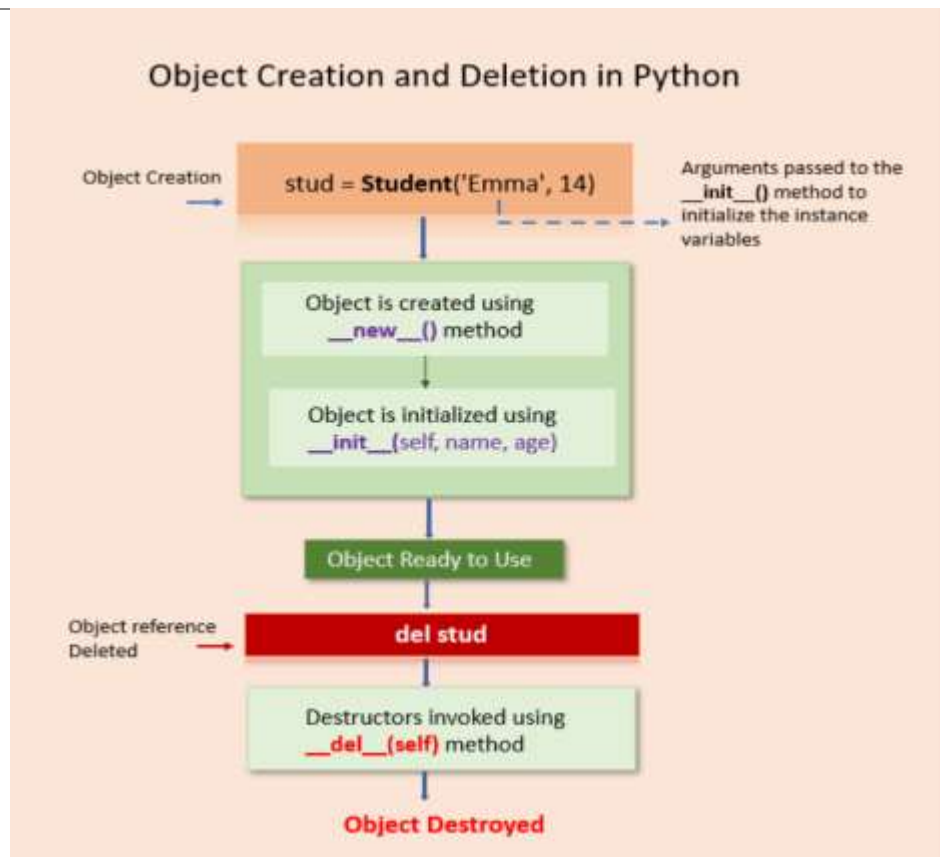


Image 41: Python – Object creation & deletion

source: <https://images.app.goo.gl/CPtf96eobgf24M3MA>

## Databases

### MySQL Databases

- MySQL is one of the most popular database management systems (DBMSs) on the market today.
- It ranked second only to the Oracle DBMS in this year's DB-Engines Ranking.
- As most software applications need to interact with data in some form, programming languages like Python provide tools for storing and accessing these data sources.

- Being open source since its inception in 1995, MySQL quickly became a market leader among SQL solutions.
- MySQL is also a part of the Oracle ecosystem.
- While its core functionality is completely free, there are some paid add-ons as well.
- Currently, MySQL is used by all major tech firms, including Google, LinkedIn, Uber, Netflix, Twitter, and others.

### Features of MySQL Database

- **Ease of installation:** MySQL was designed to be user-friendly. It's quite straightforward to set up a MySQL database, and several widely available third-party tools, like phpMyAdmin, further streamline the setup process. MySQL is available for all major operating systems, including Windows, macOS, Linux, and Solaris.
- **Speed:** MySQL holds a reputation for being an exceedingly fast database solution. It has a relatively smaller footprint and is extremely scalable in the long run.
- **User privileges and security:** MySQL comes with a script that allows you to set the password security level, assign admin passwords, and add and remove user account privileges. This script uncomplicates the admin process for a web hosting user management portal. Other DBMSs, like PostgreSQL, use config files that are more complicated to use.

### Installing MySQL Connector/Python

- A database driver is a piece of software that allows an application to connect and interact with a database system. Programming languages like Python need a special driver before they can speak to a database from a specific vendor.
- In Python you need to install a Python MySQL connector to interact with a MySQL database. Many packages follow the DB-API standards, but the most popular among them is MySQL Connector/Python. You can get it with pip:
- `pip install mysql-connector-python`
- To test if the installation was successful, type the following command on your Python terminal:
- `import mysql.connector`
- If the above code executes with no errors, then `mysql.connector` is installed and ready to use. If you encounter any errors, then make sure you're in the correct virtual environment and you're

---

using the right Python interpreter.

- Make sure that you're installing the correct `mysql-connector-python` package, which is a pure-Python implementation. Beware of similarly named but now deprecated connectors like `mysql-connector`.

## Establishing a Connection with MySQL Server

MySQL is a server-based database management system. One server might contain multiple databases. To interact with a database, you must first establish a connection with the server. The general workflow of a Python program that interacts with a MySQL-based database is as follows:

- Connect to the MySQL server.
- Create a new database.
- Connect to the newly created or an existing database.
- Execute a SQL query and fetch results.
- Inform the database if any changes are made to a table.
- Close the connection to the MySQL server.
- This is a generic workflow that might vary depending on the individual application. But whatever the application might be, the first step is to connect your database with your application.

1. The first step in interacting with a MySQL server is to establish a connection.
2. To do this, you need `connect()` from the `mysql.connector` module.
3. This function takes in parameters like `host`, `user`, and `password` and returns a `MySQLConnection` object.
4. You can receive these credentials as input from the user and pass them to `connect()`:

There are several important things to notice in the code above:

1. You should always deal with the exceptions that might be raised while establishing a connection to the MySQL server. This is why you use a `try ... except` block to catch and print any exceptions that you might encounter.
2. You should always close the connection after you're done accessing the database. Leaving

---

unused open connections can lead to several unexpected errors and performance issues.

3. You should never hard-code your login credentials, that is, your username and password, directly in a Python script. This is a bad practice for deployment and poses a serious security threat. The code above prompts the user for login credentials. It uses the built-in `getpass` module to hide the password. While this is better than hard-coding, there are other, more secure ways to store sensitive information, like using environment variables.

### Creating a new Database

- To create a new database, you need to execute a SQL statement:
- `CREATE DATABASE books_db;`
- The above statement will create a new database with the name `books_db`.
- To execute a SQL query in Python, you'll need to use a cursor, which abstracts away the access to database records.
- MySQL Connector/Python provides you with the `MySQLCursor` class, which instantiates objects that can execute MySQL queries in Python.
- An instance of the `MySQLCursor` class is also called a cursor.

### Show Database

- You might receive an error here if a database with the same name already exists in your server.
- To confirm this, you can display the name of all databases in your server.
- Using the same `MySQLConnection` object from earlier, execute the `SHOW DATABASES` statement:

```
show_db_query = "SHOW DATABASES"
with connection.cursor() as cursor:
    cursor.execute(show_db_query)
    for db in cursor:
        print(db)
```

## Creating Table:

- For creating tables we will follow the similar approach of writing the SQL commands as strings and then passing it to the execute() method of the cursor object.
- **SQL command for creating a table is –**

```
CREATE TABLE
(
    column_name_1 column_Data_type,
    column_name_2 column_Data_type,
    :
    :
    column_name_n column_Data_type
);
```

## Program for creating a Table

```
import mysql.connector
dataBase = mysql.connector.connect(
host="localhost",
user="user",
passwd="password",
database="gfg")
cursorObject = dataBase.cursor()
studentRecord = """CREATE TABLE STUDENT (
                        NAME VARCHAR(20) NOT NULL,
```

```

        BRANCH VARCHAR(50),
        ROLL INT NOT NULL,
        SECTION VARCHAR(5),
        AGE INT)""""

```

# table created

cursorObject.execute(studentRecord)

dataBase.close()

## Output

```

mysql> show tables;
+-----+
| Tables_in_gfg |
+-----+
| STUDENT       |
+-----+
1 row in set (0.01 sec)

mysql> desc STUDENT;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| NAME  | varchar(20) | NO   |     | NULL    |       |
| BRANCH | varchar(50) | YES  |     | NULL    |       |
| ROLL   | int        | NO   |     | NULL    |       |
| SECTION | varchar(5) | YES  |     | NULL    |       |
| AGE    | int        | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

```

Inserting a single row

To insert data into the MySQL table Insert into query is used.

Syntax:

```
INSERT INTO table_name (column_names) VALUES (data)
```

Program

```
# preparing a cursor object
cursorObject = DataBase.cursor()

sql = "INSERT INTO STUDENT (NAME, BRANCH, ROLL, SECTION, AGE)\
VALUES (%s, %s, %s, %s, %s)"
val = ("Ram", "CSE", "85", "B", "19")

cursorObject.execute(sql, val)
DataBase.commit()

# disconnecting from server
DataBase.close()
```

### Output

```
mysql> select * from STUDENT;
+-----+-----+-----+-----+-----+
| NAME | BRANCH | ROLL | SECTION | AGE |
+-----+-----+-----+-----+-----+
| Ram  | CSE    | 85   | B       | 19  |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> 
```

### Inserting Multiple rows

To insert multiple values at once, `executemany()` method is used. This method iterates through the sequence of parameters, passing the current parameter to the `execute` method.

```
sql = "INSERT INTO STUDENT (NAME, BRANCH, ROLL, SECTION, AGE)\nVALUES (%s, %s, %s, %s, %s)"
val = [("Nikhil", "CSE", "98", "A", "18"),
      ("Nisha", "CSE", "99", "A", "18"),
      ("Rohan", "MAE", "43", "B", "20"),
      ("Amit", "ECE", "24", "A", "21"),
      ("Anil", "MAE", "45", "B", "20"),
      ("Megha", "ECE", "55", "A", "22"),
      ("Sita", "CSE", "95", "A", "19")]

cursorObject.executemany(sql, val)
dataBase.commit()
```

## Output

```
mysql> select * from STUDENT;
+-----+-----+-----+-----+-----+
| NAME   | BRANCH | ROLL  | SECTION | AGE  |
+-----+-----+-----+-----+-----+
| Ram    | CSE    | 85    | B       | 19   |
| Nikhil | CSE    | 98    | A       | 18   |
| Nisha  | CSE    | 99    | A       | 18   |
| Rohan  | MAE    | 43    | B       | 20   |
| Amit   | ECE    | 24    | A       | 21   |
| Anil   | MAE    | 45    | B       | 20   |
| Megha  | ECE    | 55    | A       | 22   |
| Sita   | CSE    | 95    | A       | 19   |
+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```



## Fetching Data

We can use the select query on the MySQL tables

```
query = "SELECT NAME, ROLL FROM STUDENT"
cursorObject.execute(query)

myresult = cursorObject.fetchall()

for x in myresult:
    print(x)
```

## Output

```
('Ram', 85)
('Nikhil', 98)
('Nisha', 99)
('Rohan', 43)
('Amit', 24)
('Anil', 45)
('Megha', 55)
('Sita', 95)
```

## Where Clause

Where clause is used in MySQL database to filter the data as per the condition required.

You can fetch, delete or update a particular set of data in MySQL database by using where clause.

```
query = "SELECT * FROM STUDENT where AGE >=20"
cursorObject.execute(query)

myresult = cursorObject.fetchall()

for x in myresult:
    print(x)
```

## Output

```
( 'Rohan', 'MAE', 43, 'B', 20)
( 'Amit', 'ECE', 24, 'A', 21)
( 'Anil', 'MAE', 45, 'B', 20)
( 'Megha', 'ECE', 55, 'A', 22)
```

## Update Data

The update query is used to change the existing values in a database. By using update a specific value can be corrected or updated. It only affects the data and not the structure of the table. The basic advantage provided by this command is that it keeps the table accurate

```
# preparing a cursor object
cursorObject = DataBase.cursor()

query = "UPDATE STUDENT SET AGE = 23 WHERE Name ='Ram'"
cursorObject.execute(query)
DataBase.commit()
```

## Output

```
mysql> select * from STUDENT;
+-----+-----+-----+-----+-----+
| NAME | BRANCH | ROLL | SECTION | AGE |
+-----+-----+-----+-----+-----+
| Ram   | CSE    | 85   | B       | 23  |
| Nikhil | CSE    | 98   | A       | 18  |
| Nisha | CSE    | 99   | A       | 18  |
| Rohan | MAE    | 43   | B       | 20  |
| Amit  | ECE    | 24   | A       | 21  |
| Anil  | MAE    | 45   | B       | 20  |
| Megha | ECE    | 55   | A       | 22  |
| Sita  | CSE    | 95   | A       | 19  |
+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

## Delete Data from Table

We can use the Delete query to delete data from the table in MySQL.

```
query = "DELETE FROM STUDENT WHERE NAME = 'Ram'"
cursorObject.execute(query)
dataBase.commit()
```

Output

```
mysql> select * from STUDENT;
+-----+-----+-----+-----+-----+
| NAME   | BRANCH | ROLL  | SECTION | AGE  |
+-----+-----+-----+-----+-----+
| Nikhil | CSE    | 98    | A       | 18   |
| Nisha  | CSE    | 99    | A       | 18   |
| Rohan  | MAE    | 43    | B       | 20   |
| Amit   | ECE    | 24    | A       | 21   |
| Anil   | MAE    | 45    | B       | 20   |
| Megha  | ECE    | 55    | A       | 22   |
| Sita   | CSE    | 95    | A       | 19   |
+-----+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

## Drop Tables

Drop command affects the structure of the table and not data. It is used to delete an already existing table. For cases where you are not sure if the table to be dropped exists or not DROP TABLE IF EXISTS command is used

```
mysql> show tables
-> ;
+-----+
| Tables_in_gfg |
+-----+
| STUDENT      |
| Student      |
+-----+
2 rows in set (0.00 sec)
```

```
query ="DROP TABLE Student;"

cursorObject.execute(query)
dataBase.commit()
```

Output after dropping table Student

```
mysql> show tables;
+-----+
| Tables_in_gfg |
+-----+
| STUDENT      |
+-----+
1 row in set (0.00 sec)
```

### Orderby Clause

OrderBy is used to arrange the result set in either ascending or descending order.

By default, it is always in ascending order unless “DESC” is mentioned, which arranges it in descending order.

“ASC” can also be used to explicitly arrange it in ascending order. But, it is generally not done this way since default already does that.

```
query = "SELECT * FROM STUDENT ORDER BY NAME DESC"
cursorObject.execute(query)

myresult = cursorObject.fetchall()

for x in myresult:
    print(x)
```

Output

```
('Sita', 'CSE', 95, 'A', 19)
('Rohan', 'MAE', 43, 'B', 20)
('Ram', 'CSE', 85, 'B', 19)
('Nisha', 'CSE', 99, 'A', 18)
('Nikhil', 'CSE', 98, 'A', 18)
('Megha', 'ECE', 55, 'A', 22)
('Anil', 'MAE', 45, 'B', 20)
('Amit', 'ECE', 24, 'A', 21)
```

# Python Web Django and Flask

## Django

### What exactly is Django?

- Django is a Python-based web framework that allows you to easily build web applications without the installation or dependency issues that other frameworks typically have.
- A mechanism to handle user authentication (signing up, signing in, and signing out), a management panel for your website, forms, and a way to upload files are all necessary components when establishing a website.
- Django provides you with ready-to-use components.

## Why Django, exactly?

- Switching databases in the Django framework is fairly simple.
- It features a built-in admin interface that makes working with it simple.
- Django is a complete framework that requires no other software.
- There are thousands of more packages to choose from.
- It's quite adaptable.

## Django's popularity

- Many major websites utilise Django, including Disqus, Instagram, Knight Foundation, MacArthur Foundation, Mozilla, National Geographic, and others. The Django framework is used by over 5,000 websites online. (Reference)
- Hot Frameworks measures a framework's popularity by calculating the number of GitHub projects and StackOverflow inquiries for each platform; Django is ranked sixth. Web frameworks are frequently referred to be "opinionated" or "un-opinionated" depending on their views on the best approach to perform a certain task. Django is opinionated, therefore he gives you the best of both worlds ( opinionated & un-opinionated ).

### Scalability

- Django web nodes have no stored state and scale horizontally, so you can merely add more as needed. The capacity to do so is the core of scalability. Instagram and Disqus, two Django-based businesses with millions of active users, are used to demonstrate Django's scalability.

### Portability

- The Django framework is built entirely in Python, which operates on a variety of platforms. This allows Django to operate on a variety of platforms, including Linux, Windows, and Mac OS.

## Django's characteristics

## Django's versatility

Django may be used to create nearly any sort of website. It can also interact with any client-side framework and send content in HTML, JSON, XML, and other formats. Wikis, social networks, and new sites are all examples of sites that can be made with Django.

## Security

Because the Django framework is designed to make web development simple, it has been intended to perform the appropriate things to defend the website automatically. Instead of storing a password in cookies, the hashed password is saved in the Django framework, making it difficult for hackers to retrieve.

Data bases

Refer <https://www.geeksforgeeks.org/django-introduction-and-installation/?ref=lbp> for more details on installation

## FLASK

### What is Flask?

Flask is a web application framework developed in Python. Armin Ronacher, who runs an international organisation of Python fans known as Pocco, created it. The Werkzeug WSGI toolkit and the Jinja2 template engine are the foundations of Flask. Both are Pocco initiatives.

**WSGI:** The Online Server Gateway Interface (WSGI) has become the industry standard for developing Python web applications. The Web Server Gateway Interface (WSGI) is a standard for a common interface between the web server and web applications.

**Werkzeug:** It's a WSGI toolkit that handles requests, response objects, and other common tasks. This makes it possible to construct a web framework on top of it. Werkzeug is one of the foundations of the Flask framework.

---

## Jinja2

Jinja2 is a popular Python templating engine. To display dynamic web pages, a web templating system combines a template with a specific data source.

Flask is also known as a micro framework. It seeks to make an application's core basic but extensible. Flask lacks a built-in abstraction layer for database management, as well as support for form validation. Instead, Flask allows you to use extensions to add this functionality to your programme. Some of the common Flask extensions are explored later in the lesson.

Refer [https://www.tutorialspoint.com/flask/flask\\_environment.htm](https://www.tutorialspoint.com/flask/flask_environment.htm) for more details of installation

## Flask Characteristics

Here are some of Flask's key features.

- Support for unit testing is integrated.
- Request dispatching through RESTful API.
- The template engine is jinja2.
- It is based on Werkzeug toolkit.
- Secure cookies are supported (client-side sessions).
- Extensive documentation.
- Compatibility with the Google app engine.
- APIs are properly designed and cohesive
- Easily deployable in a production environment



## Python for Web-Django

In this section, we will read about:

- Web Framework, Django Introduction, Django Architecture
- Django MVC, MVT (Model View Template)
- Views and URL mapping, HttpRequest and HttpResponse , GET and POST Method
- Template, Render, Views, Context
- Template Editing
- SQL operation in Django
- Handling sessions, cookies and working with JSON and AJAX

## Web Framework, Django Introduction, Django Architecture:

### Web Framework:

- 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 on the World Wide Web.
- Web frameworks aim to automate the overhead associated with common activities performed in web development.



Image 42: Python – Web framework

source: <https://www.scnsoft.com/blog/web-application-framework>



## Introduction to Django

- Django is a Python-based free and open-source web framework that follows the model–views–template(MVT) architectural pattern.
- Django's primary goal is to ease the creation of complex, database-driven websites.
- The framework emphasizes reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of don't repeat yourself.

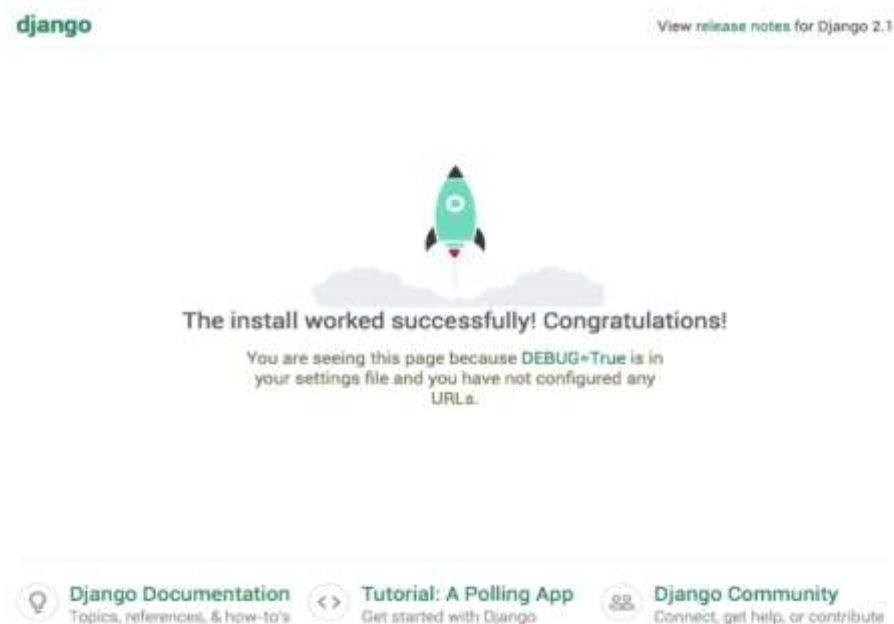


Image 43: python -Django Home Page

Source: [https://en.wikipedia.org/wiki/File:Django\\_2.1\\_landing\\_page.png](https://en.wikipedia.org/wiki/File:Django_2.1_landing_page.png)

## Django Features:

- Helps you to define patterns for the URLs in your application
- Simple but powerful URL system
- Built-in authentication system
- Object-oriented programming language database which offers best in class data storage and retrieval
- Automatic admin interface feature allows the functionality of adding, editing and deleting items. You can customize the admin panel as per your need.
- It is used for Rapid Development
- Secure
- Open Source
- Vast and Supported Community

## Django Architecture:

### 1)Model-View-Template (MVT) Architecture -

Django is based on MVT (Model-View-Template) architecture. MVT is a software design pattern for developing a web application.

- M stands for Model
- V stands for View
- T stands for Template

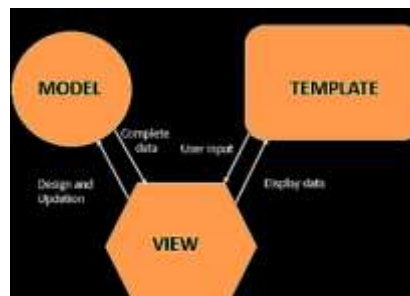


Image 44 : Python - MVT architecture

Image Source: <https://media.geeksforgeeks.org/wp-content/uploads/20210606092225/image.png>

**MVT Structure has the following three parts –**

**Model:** The model is going to act as the interface of your data. It is responsible for maintaining data. It is the logical data structure behind the entire application and is represented by a database (generally relational databases such as MySQL, Postgres).

**View:** The View is the user interface — what you see in your browser when you render a website. It is represented by HTML/CSS/Javascript and Jinja files.

**Template:** A template consists of static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

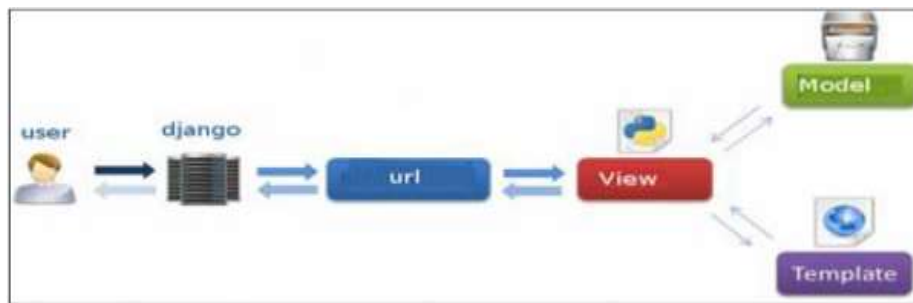


Image 45: Python - MVT Pattern

Source: [https://www.tutorialspoint.com/django/images/django\\_mvc\\_mvt\\_pattern.jpg](https://www.tutorialspoint.com/django/images/django_mvc_mvt_pattern.jpg)

## Model-View-Controller (MVC) Architecture

Model View Controller or MVC as it is popularly called, is a software design pattern for developing web applications.

A Model View Controller pattern is made up of the following three parts –

- 1)**Model** – The lowest level of the pattern which is responsible for maintaining data.
- 2)**View** – This is responsible for displaying all or a portion of the data to the user.
- 3)**Controller** – Software Code that controls the interactions between the Model and View.

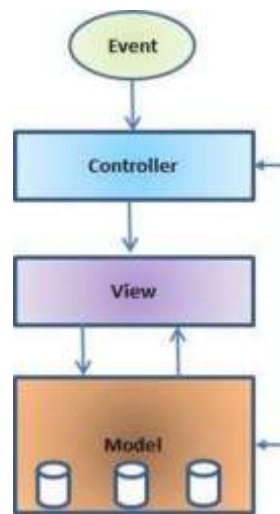


Image 46: Python - MVC Architecture

Source: [https://www.tutorialspoint.com/struts\\_2/images/struts-mvc.jpg](https://www.tutorialspoint.com/struts_2/images/struts-mvc.jpg)

MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns. Here the Controller receives all requests for the application and then works with the Model to prepare any data needed by the View. The View then uses the data prepared by the Controller to generate a final presentable response. The MVC abstraction can be graphically represented as shown in Fig1.

### The Model

The model is responsible for managing the data of the application. It responds to the request from the view and it also responds to instructions from the controller to update itself.

### The View

It means presentation of data in a particular format, triggered by a controller's decision to present the data. They are script-based templating systems like JSP, ASP, PHP and very easy to integrate with AJAX technology.

### The Controller

The controller is responsible for responding to the user input and perform interactions on the data model objects. The controller receives the input, it validates the input and then performs the business operation that modifies the state of the data model.

## Installation of Django

### Step 1) Creating environment for Django project

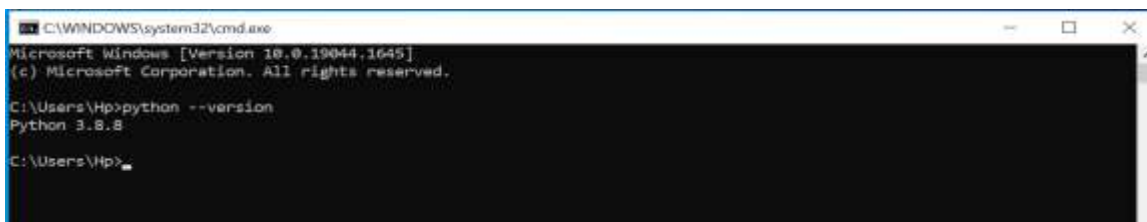
#### A) Install latest version of python

1. Download and install latest version of python from the url <https://www.python.org/downloads/>

#### B) Check for installed version of python

1. Press Window + R to open command prompt
2. Type cmd in open box and press ok button
3. Command prompt will open
4. On command prompt type following command, it will display the current python version installed on your laptop

> python --version



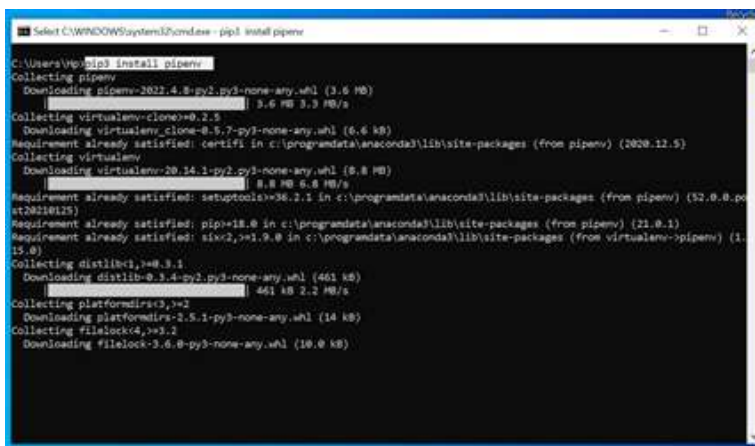
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19044.1645]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Hp>python --version
Python 3.8.8

C:\Users\Hp>
```

#### C) Install pipenv

> pip3 install pipenv

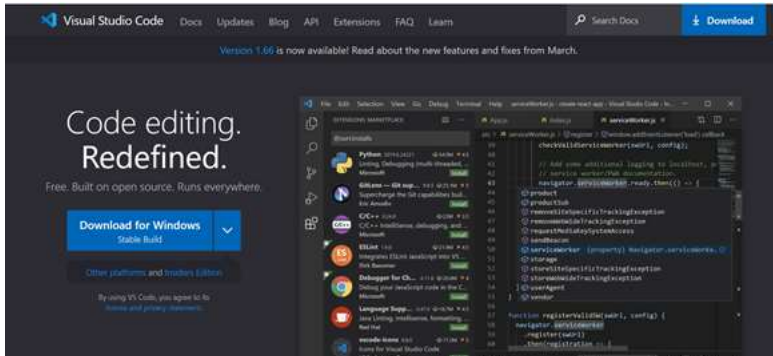


```
C:\Users\Hp>pip3 install pipenv
Collecting pipenv
  Downloading pipenv-2022.4.8-py2.py3-none-any.whl (3.6 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.6 MB 3.9 MB/s
Collecting virtualenv-clone>=0.2.5
  Downloading virtualenv_clone-0.5.7-py3-none-any.whl (6.6 kB)
Requirement already satisfied: certifi in c:\programdata\anaconda3\lib\site-packages (from pipenv) (2020.12.5)
Collecting virtualenv
  Downloading virtualenv-20.14.1-py2.py3-none-any.whl (8.8 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.8 MB 6.6 MB/s
Requirement already satisfied: setuptools>=38.2.1 in c:\programdata\anaconda3\lib\site-packages (from pipenv) (52.0.0.post20210125)
Requirement already satisfied: pip>=18.0 in c:\programdata\anaconda3\lib\site-packages (from pipenv) (21.0.1)
Requirement already satisfied: six>=1.9.0 in c:\programdata\anaconda3\lib\site-packages (from pipenv) (1.15.0)
Collecting distlib>=0.3.1
  Downloading distlib-0.3.4-py2.py3-none-any.whl (461 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 461 kB 2.2 MB/s
Collecting platformdirs>=2
  Downloading platformdirs-2.5.1-py3-none-any.whl (14 kB)
Collecting filelock>=3.2
  Downloading filelock-3.6.0-py3-none-any.whl (10.0 kB)
```



## D) Install visual studio code editor

- Visit URI and download <https://code.visualstudio.com/>



## Step 2) Creating the first Project with django

### A) Switch to Desktop

> cd Desktop

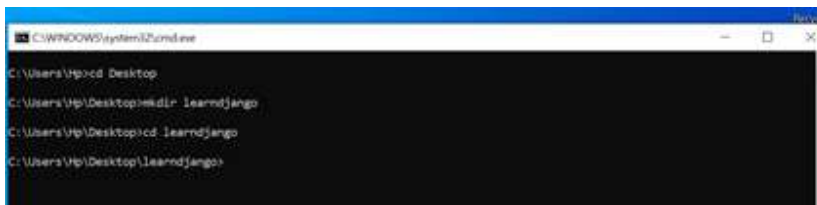
### B) Create Project folder

> mkdir <<projectname\_folder>>

> mkdir learndjango

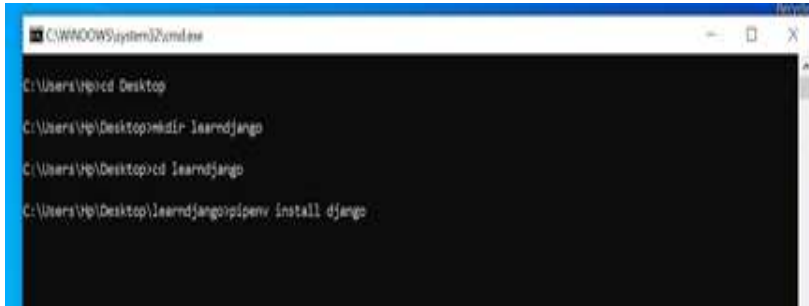
Move to learndjango directory

>cd learndjango



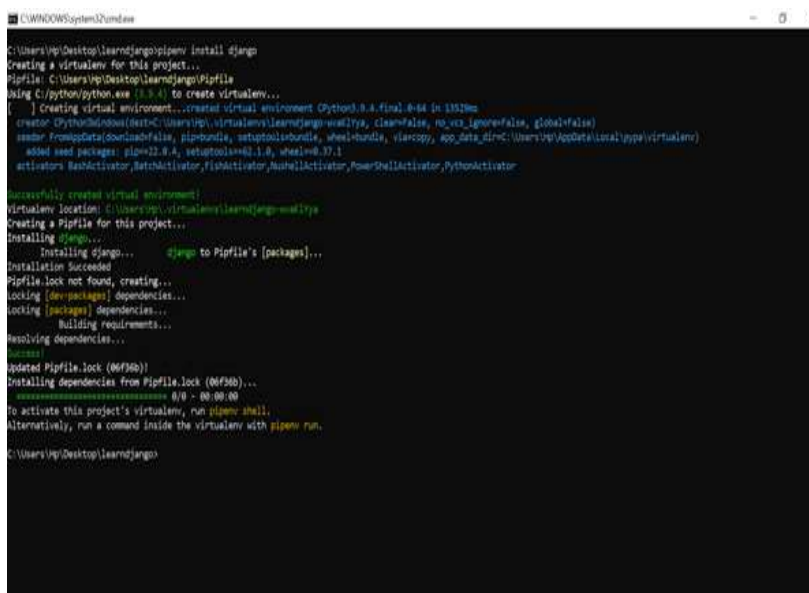
## C) Install django

>pipenv install django



```
C:\WINDOWS\system32\cmd.exe
C:\Users\Hp>cd Desktop
C:\Users\Hp\Desktop>mkdir learndjango
C:\Users\Hp\Desktop>cd learndjango
C:\Users\Hp\Desktop\learndjango>pipenv install django
```

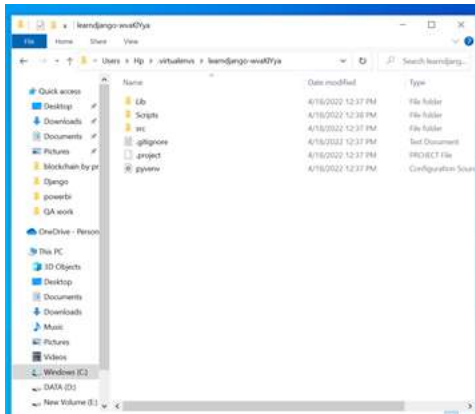
Successfully created the virtual environment



```
C:\WINDOWS\system32\cmd.exe
C:\Users\Hp\Desktop\learndjango>pipenv install django
Creating a virtualenv for this project...
Pipfile: C:\Users\Hp\Desktop\learndjango\Pipfile
Using C:\Python\python.exe (3.8.4) to create virtualenv...
[*] Creating virtual environment...created virtual environment C:\Python\python.exe (3.8.4) in 1351ms
creator: C:\Python\python.exe (3.8.4) to create virtualenv...
creator: C:\Python\python.exe (3.8.4) to create virtualenv...
added seed packages: pip==22.8.4, setuptools==68.0.0, wheel==0.37.1
activators: BashActivator, BatchActivator, FishActivator, NuShellActivator, PowerShellActivator, PythonActivator

Successfully created virtual environment!
Virtualenv location: C:\Users\Hp\Desktop\learndjango\venv
Creating a Pipfile for this project...
Installing django...
Installing django...
Installation Succeeded
Pipfile lock not found, creating...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Building requirements...
Resolving dependencies...
Success!
Updated Pipfile.lock (66f3b0)
Installing dependencies from Pipfile.lock (66f3b0)...
Success!
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.
C:\Users\Hp\Desktop\learndjango>
```

- Virtual environment location is shown in above image C:\Users\Hp\.virtualenvs\learn Django-wvaKIYya



- Activate python interpreter under this virtual environment**  
>pipenv shell



- Run django-admin to start new project.
- django admin is utility comes along with Django  
>django-admin

```
C:\WINDOWS\system32\cmd.exe - pipenv shell
(learnjango-waklvy) C:\Users\Up\Desktop\learnjango>django-admin
Type 'django-admin help -<subcommand>' for help on a specific subcommand.

Available subcommands:

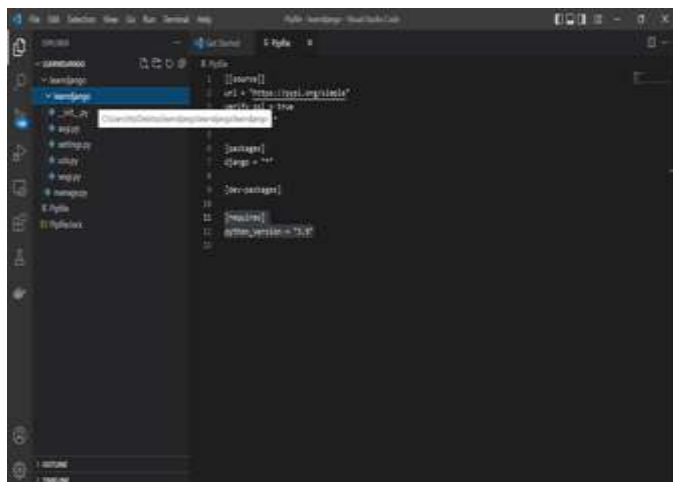
[django]
  check
  compilemessages
  createtocchetable
  dbshell
  diffsettings
  dumpdata
  flush
  inspectdb
  loaddata
  makemessages
  makemigrations
  migrate
  runserver
  sendtestemail
  shell
  showmigrations
  sqlflush
  sqlmigrate
  sqlsequencereset
  squashmigrations
  startapp
  startproject
  test
  testserver

Note that only Django core commands are listed as settings are not properly configured (error: Requested setting INSTALLED_APPS, but settings are not configured. You must
either define the environment variable DJANGO_SETTINGS_MODULE or call settings.configure() before accessing settings.).

(learnjango-waklvy) C:\Users\Up\Desktop\learnjango>
```

## D) start our project learndjango

```
C:\WINDOWS\system32\cmd.exe - powershell
(learnjango-wak1ya) C:\Users\Up\Desktop\learnjango>django-admin startproject learnjango
(learnjango-wak1ya) C:\Users\Up\Desktop\learnjango>
```



- It creates the two directories with the same name **learndjango**

-First directory learndjango is the project directory and second directory learndjango is the django application directory

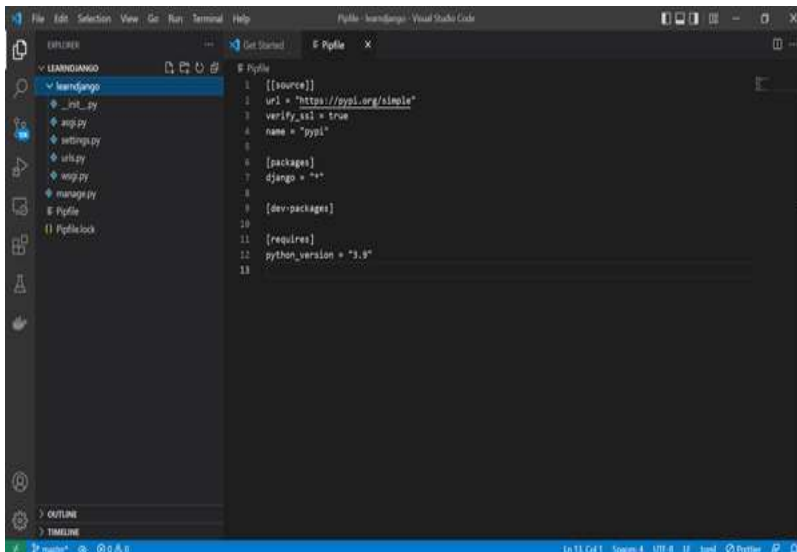
-delete the first directory learndjango and go to the command prompt

### E) Go Back to the terminal and execute the command

> django-admin startproject learndjango



```
C:\WINDOWS\system32\cmd.exe - powershell
(learndjango-wsk17ya) C:\Users\vip\Desktop\learndjango>django-admin startproject learndjango
(learndjango-wsk17ya) C:\Users\vip\Desktop\learndjango>django-admin startproject learndjango .
(learndjango-wsk17ya) C:\Users\vip\Desktop\learndjango>
```



It will use current directory as our project directory .It will now not going to create the additional directory.

### F) Start webserver

- To Start webserver - Run the command

>python manage.py runserver

It will start server at <http://127.0.0.1:8000/>

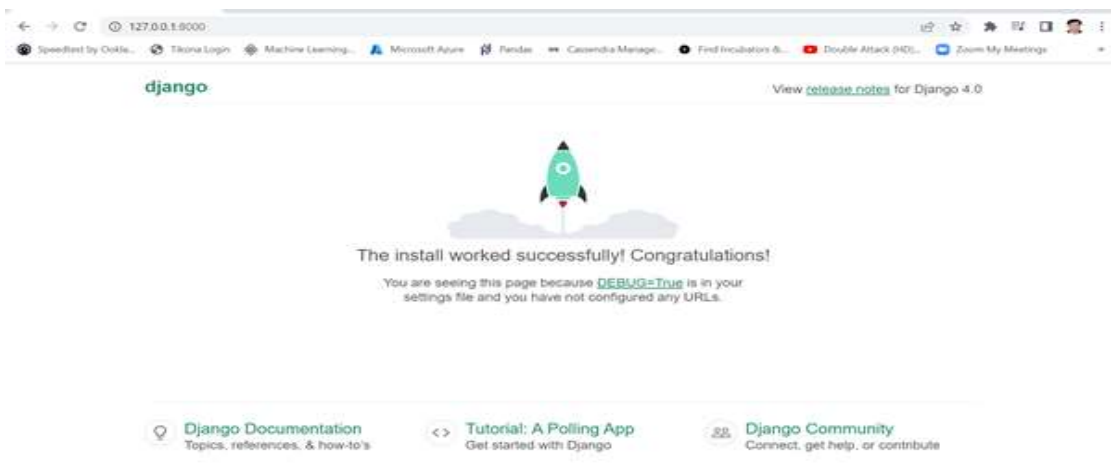
```
CSWINDOWS\system32\cmd.exe - pipenv shell - python manage.py runserver
(learndjango-uvx1vya) C:\Users\ap\Desktop\learndjango>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 18, 2022 - 13:59:01
Django version 4.0.4, using settings 'learndjango.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

### G) Load the landing page/welcome page of django

- Now open browser and type address <http://127.0.0.1:8000/> to open home page of our Django project learndjango



**Congratulation! You have installed and run the Django home page successfully**

# Writing your first Django app: basic poll application (Part1)

Throughout this tutorial, we'll walk you through the creation of a basic poll application.

It'll consist of two parts:

- A public site that lets people view polls and vote in them.
- An admin site that lets you add, change, and delete polls.

We'll assume you have [Django installed](#) already. You can tell Django is installed and which version by running the following command in a shell prompt (indicated by the \$ prefix):

```
$ python -m django --version
```

If Django is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named django".

This tutorial is written for Django 4.0, which supports Python 3.8 and later. If the Django version doesn't match, you can refer to the tutorial for your version of Django by using the version switcher at the bottom right corner of this page, or update Django to the newest version. If you're using an older version of Python, check [What Python version can I use with Django?](#) to find a compatible version of Django.

See [How to install Django](#) for advice on how to remove older versions of Django and install a newer one.

## Creating a 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. If it didn't work, see [Problems running django-admin](#).

### Note

You'll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like **django** (which will conflict with Django itself) or **test** (which conflicts with a built-in Python package).

### Where should this code live?

If your background is in plain old PHP (with no use of modern frameworks), you're probably used to putting code under the web server's document root (in a place such as **/var/www**). With Django, you don't do that. It's not a good idea to put any of this Python code within your web server's document root, because it risks the possibility that people may be able to view your code over the web. That's not good for security.

Put your code in some directory **outside** of the document root, such as **/home/mycode**.

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



```
mysite/

manage.py

mysite/

    __init__.py

    settings.py

    urls.py

    asgi.py

    wsgi.py
```

These files are:

- The outer **mysite/** root directory is 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. You can read all the details about **manage.py** in [django-admin and manage.py](#).
- 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. If you're a Python beginner, read [more about packages](#) in the official Python docs.
- **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. You can read more about URLs in [URL dispatcher](#).

- **mysite/asgi.py**: An entry-point for ASGI-compatible web servers to serve your project. See [How to deploy with ASGI](#) for more details.
- **mysite/wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project. See [How to deploy with WSGI](#) for more details.

## 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.
```

```
April 29, 2022 - 15:50:53
```

```
Django version 4.0, using settings 'mysite.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

### Note

Ignore the warning about unapplied database migrations for now; we'll deal with the database shortly.

You've started the Django development server, a lightweight web server written purely in Python. We've included this with Django so you can develop things rapidly, without having to deal with configuring a production server – such as Apache – until you're ready for production.

Now's a good time to note: **don't** use this server in anything resembling a production environment. It's intended only for use while developing. (We're in the business of making web frameworks, not web servers.)

Now that the server's running, visit <http://127.0.0.1:8000/> with your web browser. You'll see a "Congratulations!" page, with a rocket taking off. It worked!

## Changing the port

By default, the **runserver** command starts the development server on the internal IP at port 8000.

If you want to change the server's port, pass it as a command-line argument. For instance, this command starts the server on port 8080:

```
$ python manage.py runserver 8080
```

If you want to change the server's IP, pass it along with the port. For example, to listen on all available public IPs (which is useful if you are running Vagrant or want to show off your work on other computers on the network), use:

```
$ python manage.py runserver 0:8000
```

**0** is a shortcut for **0.0.0.0**. Full docs for the development server can be found in the **runserver** reference.

## Automatic reloading of runserver

The development server automatically reloads Python code for each request as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

---

## Creating the Polls app

Now that your environment – a “project” – is set up, you’re set to start doing work.

Each application you write in Django consists of a Python package that follows a certain convention. Django comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories.

### Projects vs. apps

What’s the difference between a project and an app? An app is a web application that does something – e.g., a blog system, a database of public records or a small poll app. A project is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.

Your apps can live anywhere on your **Python path**. In this tutorial, we’ll create our poll app in the same directory as your **manage.py** file so that it can be imported as its own top-level module, rather than a submodule of **mysite**.

To create your app, make sure you’re in the same directory as **manage.py** and type this command:

```
$ python manage.py startapp polls
```

That’ll create a directory **polls**, which is laid out like this:

```
polls/
  __init__.py
  admin.py
  apps.py
  migrations/
    __init__.py
```

```
models.py
```

```
tests.py
```

```
views.py
```

This directory structure will house the poll application.

## Write your first view

Let's write the first view. Open the file **polls/views.py** and put the following Python code in it:

```
polls/views.py
```

```
from django.http import HttpResponse

def index(request):

    return HttpResponse("Hello, world. You're at the polls index.")
```

This is the simplest view possible in Django. To call the view, we need to map it to a URL - and for this we need a URLconf.

To create a URLconf in the polls directory, create a file called **urls.py**. Your app directory should now look like:

```
polls/

__init__.py

admin.py

apps.py
```

```
migrations/

__init__.py

models.py

tests.py

urls.py

views.py
```

In the **polls/urls.py** file include the following code:

polls/urls.py

```
from django.urls import path

from . import views

urlpatterns = [

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

]
```

The next step is to point the root URLconf at the **polls.urls** module. In **mysite/urls.py**, add an import for **django.urls.include** and insert an **include()** in the **urlpatterns** list, so you have:

mysite/urls.py

```
from django.contrib import admin

from django.urls import include, path
```

```
urlpatterns = [  
    path('polls/', include('polls.urls')),  
    path('admin/', admin.site.urls),  
]
```

The **include()** function allows referencing other URLconfs. Whenever Django encounters **include()**, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

The idea behind **include()** is to make it easy to plug-and-play URLs. Since polls are in their own URLconf (**polls/urls.py**), they can be placed under “/polls/”, or under “/fun\_polls/”, or under “/content/polls/”, or any other path root, and the app will still work.

### When to use include()

You should always use **include()** when you include other URL patterns. **admin.site.urls** is the only exception to this.

You have now wired an **index** view into the URLconf. Verify it’s working with the following command:

```
$ python manage.py runserver
```

Go to <http://localhost:8000/polls/> in your browser, and you should see the text “*Hello, world. You’re at the polls index.*”, which you defined in the **index** view.

### Page not found?

If you get an error page here, check that you’re going to <http://localhost:8000/polls/> and not <http://localhost:8000/>.

The **path()** function is passed four arguments, two required: **route** and **view**, and two optional: **kwargs**, and **name**. At this point, it's worth reviewing what these arguments are for.

### **path() argument: route**

**route** is a string that contains a URL pattern. When processing a request, Django starts at the first pattern in **urlpatterns** and makes its way down the list, comparing the requested URL against each pattern until it finds one that matches.

Patterns don't search GET and POST parameters, or the domain name. For example, in a request to **https://www.example.com/myapp/**, the URLconf will look for **myapp/**. In a request to **https://www.example.com/myapp/?page=3**, the URLconf will also look for **myapp/**.

### **path() argument: view**

When Django finds a matching pattern, it calls the specified view function with an **HttpRequest** object as the first argument and any "captured" values from the route as keyword arguments. We'll give an example of this in a bit.

### **path() argument: kwargs**

Arbitrary keyword arguments can be passed in a dictionary to the target view. We aren't going to use this feature of Django in the tutorial.

### **path() argument: name**

Naming your URL lets you refer to it unambiguously from elsewhere in Django, especially from within templates. This powerful feature allows you to make global changes to the URL patterns of your project while only touching a single file.





## Writing your first Django app, part 2

**Now we will add database in our Project -**

We'll set up the database, create your first model, and get a quick introduction to Django's automatically-generated admin site.

### Where to get help:

If you're having trouble going through this tutorial, please head over to the [Getting Help](#) section of the FAQ.

### Database setup

Now, 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.

If you wish to use another database, install the appropriate [database bindings](#) and change the following keys in the **DATABASES 'default'** item to match your database connection settings:

- **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, **BASE\_DIR / 'db.sqlite3'**, will store the file in your project directory.

If you are not using SQLite as your database, additional settings such as **USER**, **PASSWORD**, and **HOST** must be added. For more details, see the reference documentation for **DATABASES**.

### For databases other than SQLite

If you're using a database besides SQLite, make sure you've created a database by this point. Do that with "**CREATE DATABASE database\_name;**" within your database's interactive prompt.

Also make sure that the database user provided in **mysite/settings.py** has "create database" privileges. This allows automatic creation of a **test database** which will be needed in a later tutorial.

If you're using SQLite, you don't need to create anything beforehand - the database file will be created automatically when it is needed.

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
```

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. If you're interested, run the command-line client for your database and type **\dt** (PostgreSQL), **SHOW TABLES;** (MariaDB, MySQL), **.tables** (SQLite), or **SELECT TABLE\_NAME FROM USER\_TABLES;** (Oracle) to display the tables Django created.

### For the minimalists

Like we said above, the default applications are included for the common case, but not everybody needs them. If you don't need any or all of them, feel free to comment-out or delete the appropriate line(s) from **INSTALLED\_APPS** before running **migrate**. The **migrate** command will only run migrations for apps in **INSTALLED\_APPS**.

## Creating models

Now we'll define your models – essentially, your database layout, with additional metadata.

### Philosophy

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. Django follows the DRY Principle. The goal is to define your data model in one place and automatically derive things from it.

This includes the migrations - unlike in Ruby On Rails, for example, migrations are entirely derived from your models file, and are essentially a history that Django can roll through to update your database schema to match your current models.



In our 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 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)
```

Here, 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.

## Activating models

That small bit of model code gives Django a lot of information. With it, Django is able to:

- Create a database schema (**CREATE TABLE** statements) for this app.
- Create a Python database-access API for accessing **Question** and **Choice** objects.

But first we need to tell our project that the **polls** app is installed.

## Philosophy

Django apps are “pluggable”: You can use an app in multiple projects, and you can distribute apps, because they don't have to be tied to a given Django installation.

To include the app in our project, we need to add a reference to its configuration class in the **INSTALLED\_APPS** setting. The **PollsConfig** class is in the **polls/apps.py** file, so its dotted path is **'polls.apps.PollsConfig'**. Edit the **mysite/settings.py** file and add that dotted path to the **INSTALLED\_APPS** setting. It'll look like this:

mysite/settings.py

```
INSTALLED_APPS = [  
    'polls.apps.PollsConfig',  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
]
```

Now Django knows to include the **polls** app. Let's run another command:

```
$ python manage.py makemigrations polls
```

You should see something similar to the following:

```
Migrations for 'polls':  
  
polls/migrations/0001_initial.py  
  
- Create model Question
```



#### - Create model 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 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.

There's a command that will run the migrations for you and manage your database schema automatically - that's called **migrate**, and we'll come to it in a moment - but first, let's see what SQL that migration would run. The **sqlmigrate** command takes migration names and returns their SQL:

```
$ python manage.py sqlmigrate polls 0001
```

You should see something similar to the following (we've reformatted it for readability):

```
BEGIN;

--

-- Create model Question

--

CREATE TABLE "polls_question" (

    "id" serial NOT NULL PRIMARY KEY,

    "question_text" varchar(200) NOT NULL,

    "pub_date" timestamp with time zone NOT NULL

);
```



```
--
-- Create model Choice
--

CREATE TABLE "polls_choice" (

    "id" serial NOT NULL PRIMARY KEY,

    "choice_text" varchar(200) NOT NULL,

    "votes" integer NOT NULL,

    "question_id" integer NOT NULL

);

ALTER TABLE "polls_choice"

    ADD CONSTRAINT "polls_choice_question_id_c5b4b260_fk_polls_question_id"

        FOREIGN KEY ("question_id")

        REFERENCES "polls_question" ("id")

        DEFERRABLE INITIALLY DEFERRED;

CREATE INDEX "polls_choice_question_id_c5b4b260" ON "polls_choice" ("question_id");

COMMIT;
```

Note the following:

- The exact output will vary depending on the database you are using. The example above is generated for PostgreSQL.



- Table names are automatically generated by combining the name of the app (**polls**) and the lowercase name of the model – **question** and **choice**. (You can override this behavior.)
- Primary keys (IDs) are added automatically. (You can override this, too.)
- By convention, Django appends "**\_id**" to the foreign key field name. (Yes, you can override this, as well.)
- The foreign key relationship is made explicit by a **FOREIGN KEY** constraint. Don't worry about the **DEFERRABLE** parts; it's telling PostgreSQL to not enforce the foreign key until the end of the transaction.
- It's tailored to the database you're using, so database-specific field types such as **auto\_increment** (MySQL), **serial** (PostgreSQL), or **integer primary key autoincrement** (SQLite) are handled for you automatically. Same goes for the quoting of field names – e.g., using double quotes or single quotes.
- The **sqlmigrate** command doesn't actually run the migration on your database - instead, it prints it to the screen so that you can see what SQL Django thinks is required. It's useful for checking what Django is going to do or if you have database administrators who require SQL scripts for changes.

If you're interested, you can also run **python manage.py check**; this checks for any problems in your project without making migrations or touching the database.

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 usable by other developers and in production.

Read the [django-admin documentation](#) for full information on what the **manage.py** utility can do.

## Playing with the API

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.

Once you're in the shell, explore the database API:

```
>>> from polls.models import Choice, Question # Import the model classes we just wrote.

# No questions are in the system yet.

>>> Question.objects.all()

<QuerySet []>

# Create a new Question.

# Support for time zones is enabled in the default settings file, so

# Django expects a datetime with tzinfo for pub_date. Use timezone.now()

# instead of datetime.datetime.now() and it will do the right thing.

>>> from django.utils import timezone

>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.

>>> q.save()

# Now it has an ID.

>>> q.id
```

1

```
# Access model field values via Python attributes.
```

```
>>> q.question_text
```

```
"What's new?"
```

```
>>> q.pub_date
```

```
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)
```

```
# Change values by changing the attributes, then calling save().
```

```
>>> q.question_text = "What's up?"
```

```
>>> q.save()
```

```
# objects.all() displays all the questions in the database.
```

```
>>> Question.objects.all()
```

```
<QuerySet [<Question: Question object (1)>]>
```

Wait a minute. **<Question: Question object (1)>** isn't a helpful representation of this object. Let's fix that by editing the **Question** model (in the **polls/models.py** file) and adding a **\_\_str\_\_()** method to both **Question** and **Choice**:

polls/models.py1

```
from django.db import models
```

```
class Question(models.Model):
```

```
    # ...
```

```
    def __str__(self):
```

```
        return self.question_text
```

```
class Choice(models.Model):
```

```
    # ...
```

```
    def __str__(self):
```

```
        return self.choice_text
```

It's important to add `__str__()` methods to your models, not only for your own convenience when dealing with the interactive prompt, but also because objects' representations are used throughout Django's automatically-generated admin.

Let's also add a custom method to this model:

polls/models.py

```
import datetime
```

```
from django.db import models
```

```
from django.utils import timezone
```

```
class Question(models.Model):  
  
    # ...  
  
    def was_published_recently(self):  
  
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Note the addition of **import datetime** and **from django.utils import timezone**, to reference Python's standard **datetime** module and Django's time-zone-related utilities in **django.utils.timezone**, respectively. If you aren't familiar with time zone handling in Python, you can learn more in the [time zone support docs](#).

Save these changes and start a new Python interactive shell by running **python manage.py shell** again:

```
>>> from polls.models import Choice, Question  
  
# Make sure our __str__() addition worked.  
  
>>> Question.objects.all()  
  
<QuerySet [<Question: What's up?>]>  
  
# Django provides a rich database lookup API that's entirely driven by  
# keyword arguments.  
  
>>> Question.objects.filter(id=1)  
  
<QuerySet [<Question: What's up?>]>
```

```
>>> Question.objects.filter(question__text__startswith='What')
```

```
<QuerySet [<Question: What's up?>]>
```

```
# Get the question that was published this year.
```

```
>>> from django.utils import timezone
```

```
>>> current_year = timezone.now().year
```

```
>>> Question.objects.get(pub_date__year=current_year)
```

```
<Question: What's up?>
```

```
# Request an ID that doesn't exist, this will raise an exception.
```

```
>>> Question.objects.get(id=2)
```

```
Traceback (most recent call last):
```

```
...
```

```
DoesNotExist: Question matching query does not exist.
```

```
# Lookup by a primary key is the most common case, so Django provides a
```

```
# shortcut for primary-key exact lookups.
```

```
# The following is identical to Question.objects.get(id=1).
```

```
>>> Question.objects.get(pk=1)
```



<Question: What's up?>

```
# Make sure our custom method worked.
```

```
>>> q = Question.objects.get(pk=1)
```

```
>>> q.was_published_recently()
```

```
True
```

```
# Give the Question a couple of Choices. The create call constructs a new
```

```
# Choice object, does the INSERT statement, adds the choice to the set
```

```
# of available choices and returns the new Choice object. Django creates
```

```
# a set to hold the "other side" of a ForeignKey relation
```

```
# (e.g. a question's choice) which can be accessed via the API.
```

```
>>> q = Question.objects.get(pk=1)
```

```
# Display any choices from the related object set -- none so far.
```

```
>>> q.choice_set.all()
```

```
<QuerySet []>
```

```
# Create three choices.
```

```
>>> q.choice_set.create(choice_text='Not much', votes=0)

<Choice: Not much>

>>> q.choice_set.create(choice_text='The sky', votes=0)

<Choice: The sky>

>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)


# Choice objects have API access to their related Question objects.

>>> c.question

<Question: What's up?>


# And vice versa: Question objects get access to Choice objects.

>>> q.choice_set.all()

<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

>>> q.choice_set.count()

3


# The API automatically follows relationships as far as you need.

# Use double underscores to separate relationships.

# This works as many levels deep as you want; there's no limit.
```

```
# Find all Choices for any question whose pub_date is in this year

# (reusing the 'current_year' variable we created above).

>>> Choice.objects.filter(question__pub_date__year=current_year)

<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>


# Let's delete one of the choices. Use delete() for that.

>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')

>>> c.delete()
```

For more information on model relations, see [Accessing related objects](#). For more on how to use double underscores to perform field lookups via the API, see [Field lookups](#). For full details on the database API, see our [Database API reference](#).

## Introducing the Django Admin

### Philosophy

Generating admin sites for your staff or clients to add, change, and delete content is tedious work that doesn't require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.

Django was written in a newsroom environment, with a very clear separation between “content publishers” and the “public” site. Site managers use the system to add news stories, events, sports

scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.

The admin isn't intended to be used by site visitors. It's for site managers.

## Creating an admin user

First we'll need to create a user who can login to the admin site. Run the following command:

```
$ python manage.py createsuperuser
```

Enter your desired username and press enter.

```
Username: admin
```

You will then be prompted for your desired email address:

```
Email address: admin@example.com
```

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

```
Password: *****
```

```
Password (again): *****
```

```
Superuser created successfully.
```

## Start the development server

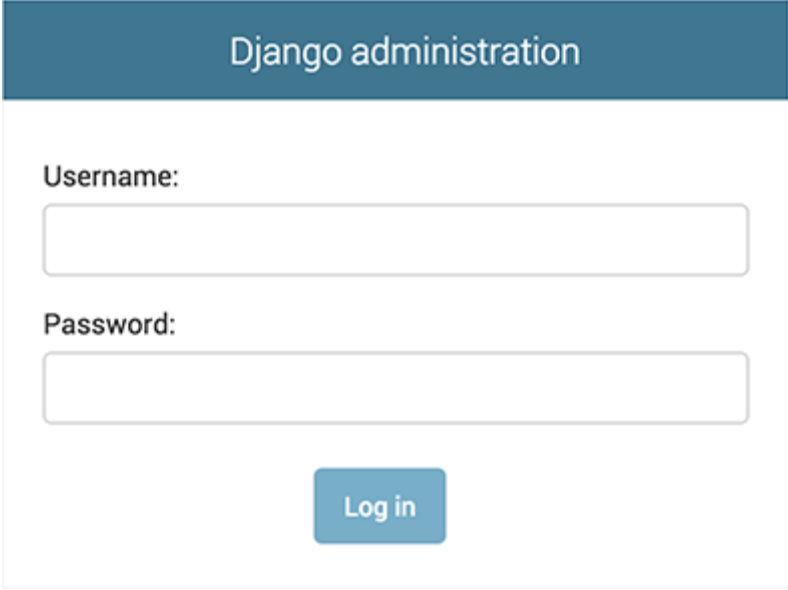
The Django admin site is activated by default. Let's start the development server and explore it.

If the server is not running start it like so:

```
$ python manage.py runserver
```

Now, open a web browser and go to “/admin/” on your local domain – e.g., <http://127.0.0.1:8000/admin/>.

You should see the admin’s login screen:

The image shows the Django administration login interface. It features a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". The "Username:" field is a simple text input. The "Password:" field is a text input with a small eye icon on the right side to toggle visibility. Below these fields is a blue button labeled "Log in".

Since **translation** is turned on by default, if you set **LANGUAGE\_CODE**, the login screen will be displayed in the given language (if Django has appropriate translations).

## Enter the admin site

Now, try logging in with the superuser account you created in the previous step. You should see the Django admin index page:

## Django administration

WELCOME, **ADMIN**. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

## Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	<a href="#">+ Add</a>	<a href="#">Change</a>
Users	<a href="#">+ Add</a>	<a href="#">Change</a>

**Recent Actions**  
  
**My Actions**  
None available

You should see a few types of editable content: groups and users. They are provided by **django.contrib.auth**, the authentication framework shipped by Django.

## Make the poll app modifiable in the admin

But where's our poll app? It's not displayed on the admin index page.

Only one more thing to do: we need to tell the admin that **Question** objects have an admin interface. To do this, open the **polls/admin.py** file, and edit it to look like this:

polls/admin.py

```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

## Explore the free admin functionality

Now that we've registered **Question**, Django knows that it should be displayed on the admin index page:



## Site administration

### AUTHENTICATION AND AUTHORIZATION

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

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

### POLLS

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

### Recent Actions

#### My Actions

None available

Click “Questions”. Now you’re at the “change list” page for questions. This page displays all the questions in the database and lets you choose one to change it. There’s the “What’s up?” question we created earlier:

[Home](#) > [Polls](#) > [Questions](#)

Select question to change

[ADD QUESTION](#) +

Action:   0 of 1 selected

☐ [QUESTION](#)

☐ [What's up?](#)

1 question

Click the “What’s up?” question to edit it:

Change question

HISTORY

Question text:

What's up?

Date published:

Date:

2015-09-06

Today



Time:

21:16:22

Now



Delete

Save and add another

Save and continue editing

SAVE

Things to note here:

- The form is automatically generated from the **Question** model.
- The different model field types (**DateTimeField**, **CharField**) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
- Each **DateTimeField** gets free JavaScript shortcuts. Dates get a “Today” shortcut and calendar popup, and times get a “Now” shortcut and a convenient popup that lists commonly entered times.

The bottom part of the page gives you a couple of options:

- Save – Saves changes and returns to the change-list page for this type of object.
- Save and continue editing – Saves changes and reloads the admin page for this object.
- Save and add another – Saves changes and loads a new, blank form for this type of object.
- Delete – Displays a delete confirmation page.



If the value of “Date published” doesn’t match the time when you created the question in **Tutorial 1**, it probably means you forgot to set the correct value for the **TIME\_ZONE** setting. Change it, reload the page and check that the correct value appears.

Change the “Date published” by clicking the “Today” and “Now” shortcuts. Then click “Save and continue editing.” Then click “History” in the upper right. You’ll see a page listing all changes made to this object via the Django admin, with the timestamp and username of the person who made the change:

Home › Polls › Questions › What's up? › History		
Change history: What's up?		
DATE/TIME	USER	ACTION
Sept. 6, 2015, 9:21 p.m.	elky	Changed pub_date.

## Views and URL mapping, HttpRequest & HttpResponse, GET & POST Method

### Views-

- A view function, or view for short, is 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 that a web browser can display.
- The convention is to put views in a file called views.py file in your project or application directory.

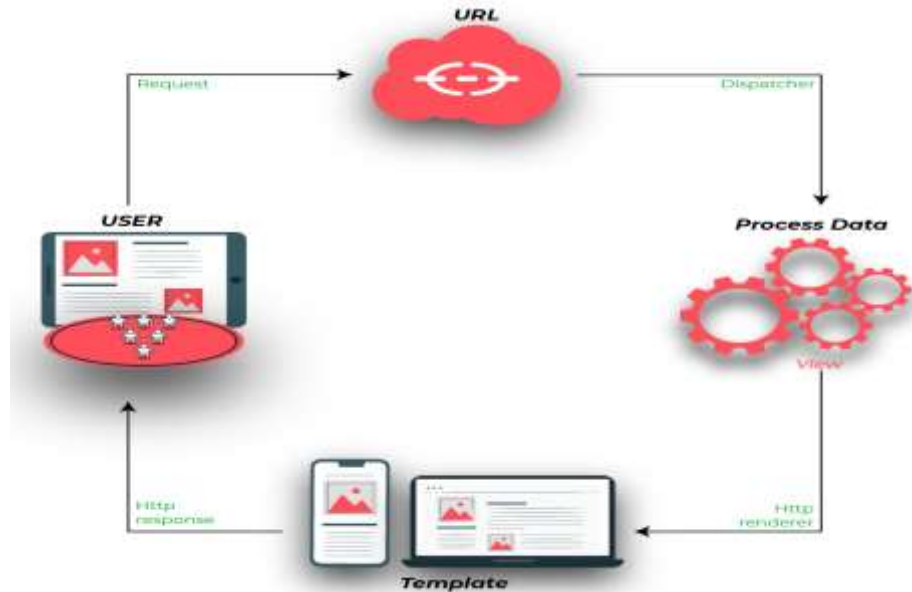


Image 47: Python - View

Source: <https://media.geeksforgeeks.org/wp-content/uploads/20200124153519/django-views.jpg>

### Creating simple View : Example -

Here's a view that returns the current date and time, as an HTML document: `learndjango/ views.py`

```
# import Http Response from django
from django.http import HttpResponse
# get datetime
import datetime
# create a function
def date_view(request):
    # fetch date and time
    now = datetime.datetime.now()
    # convert to string
    html = "Time is {}".format(now)
    # return response
    return HttpResponse(html)
```

First, we import the class `HttpResponse` from the `django.http` module, along with Python's `datetime` library.

Next, we define a function called `date_view`. This is the view function. Each view function takes an `HttpRequest` object as its first parameter, which is typically named `request`.

The view returns an `HttpResponse` object that contains the generated response. Each view function is responsible for returning an `HttpResponse` object.

### URL Mapping : Example

Let's get this view to working, in `learndjango/urls.py`

```
from django.contrib import admin
from django.urls import path, include
from .views import date_view
urlpatterns = [
    #path('admin/', admin.site.urls),
    path("", date_view),
]
```

### Output for DateTime example

Now, visit <http://127.0.0.1:8000/>



---

# Request and response objects-

## Quick overview

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.

This document explains the APIs for **HttpRequest** and **HttpResponse** objects, which are defined in the **django.http** module.

## HttpRequest objects

**class HttpRequest**

### Attributes

All attributes should be considered read-only, unless stated otherwise.

#### **HttpRequest.scheme**

A string representing the scheme of the request (**http** or **https** usually).

#### **HttpRequest.body**

The raw HTTP request body as a bytestring. This is useful for processing data in different ways than conventional HTML forms: binary images, XML payload etc. For processing conventional form data, use **HttpRequest.POST**.

You can also read from an **HttpRequest** using a file-like interface with **HttpRequest.read()** or **HttpRequest.readline()**. Accessing the **body** attribute *after* reading the request with either of these I/O stream methods will produce a **RawPostDataException**.

## HttpRequest.path

A string representing the full path to the requested page, not including the scheme, domain, or query string.

Example: `"/music/bands/the_beatles/"`

## HttpRequest.path\_info

Under some web server configurations, the portion of the URL after the host name is split up into a script prefix portion and a path info portion. The **path\_info** attribute always contains the path info portion of the path, no matter what web server is being used. Using this instead of **path** can make your code easier to move between test and deployment servers.

For example, if the **WSGIScriptAlias** for your application is set to `"/minfo"`, then **path** might be `"/minfo/music/bands/the_beatles/"` and **path\_info** would be `"/music/bands/the_beatles/"`.

## HttpRequest.method

A string representing the HTTP method used in the request. This is guaranteed to be uppercase. For example:

```
if request.method == 'GET':  
    do_something()  
  
elif request.method == 'POST':  
    do_something_else()
```

## HttpRequest.encoding

A string representing the current encoding used to decode form submission data (or **None**, which means the **DEFAULT\_CHARSET** setting is used). You can write to this attribute to change the

encoding used when accessing the form data. Any subsequent attribute accesses (such as reading from **GET** or **POST**) will use the new **encoding** value. Useful if you know the form data is not in the **DEFAULT\_CHARSET** encoding.

### **HttpRequest.content\_type**

A string representing the MIME type of the request, parsed from the **CONTENT\_TYPE** header.

### **HttpRequest.content\_params**

A dictionary of key/value parameters included in the **CONTENT\_TYPE** header.

### **HttpRequest.GET**

A dictionary-like object containing all given HTTP GET parameters. See the **QueryDict** documentation below.

### **HttpRequest.POST**

A dictionary-like object containing all given HTTP POST parameters, providing that the request contains form data. See the **QueryDict** documentation below. If you need to access raw or non-form data posted in the request, access this through the **HttpRequest.body** attribute instead.

It's possible that a request can come in via POST with an empty **POST** dictionary – if, say, a form is requested via the POST HTTP method but does not include form data. Therefore, you shouldn't use **if request.POST** to check for use of the POST method; instead, use **if request.method == "POST"** (see **HttpRequest.method**).

**POST** does *not* include file-upload information. See **FILES**.

### **HttpRequest.COOKIES**

A dictionary containing all cookies. Keys and values are strings.

### **HttpRequest.FILES**

— A dictionary-like object containing all uploaded files. Each key in **FILES** is the **name** from the `<input type="file" name="">`. Each value in **FILES** is an **UploadedFile**.

See Managing files for more information.

**FILES** will only contain data if the request method was POST and the **<form>** that posted to the request had **enctype="multipart/form-data"**. Otherwise, **FILES** will be a blank dictionary-like object.

## HttpRequest.META

A dictionary containing all available HTTP headers. Available headers depend on the client and server, but here are some examples:

- **CONTENT\_LENGTH** – The length of the request body (as a string).
- **CONTENT\_TYPE** – The MIME type of the request body.
- **HTTP\_ACCEPT** – Acceptable content types for the response.
- **HTTP\_ACCEPT\_ENCODING** – Acceptable encodings for the response.
- **HTTP\_ACCEPT\_LANGUAGE** – Acceptable languages for the response.
- **HTTP\_HOST** – The HTTP Host header sent by the client.
- **HTTP\_REFERER** – The referring page, if any.
- **HTTP\_USER\_AGENT** – The client's user-agent string.
- 
- **QUERY\_STRING** – The query string, as a single (unparsed) string.
- **REMOTE\_ADDR** – The IP address of the client.
- **REMOTE\_HOST** – The hostname of the client.
- **REMOTE\_USER** – The user authenticated by the web server, if any.

- **REQUEST\_METHOD** – A string such as "**GET**" or "**POST**".
- **SERVER\_NAME** – The hostname of the server.
- **SERVER\_PORT** – The port of the server (as a string).

With the exception of **CONTENT\_LENGTH** and **CONTENT\_TYPE**, as given above, any HTTP headers in the request are converted to **META** keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an **HTTP\_** prefix to the name. So, for example, a header called **X-Bender** would be mapped to the **META** key **HTTP\_X\_BENDER**.

Note that **runserver** strips all headers with underscores in the name, so you won't see them in **META**. This prevents header-spoofing based on ambiguity between underscores and dashes both being normalizing to underscores in WSGI environment variables. It matches the behavior of web servers like Nginx and Apache 2.4+.

**HttpRequest.headers** is a simpler way to access all HTTP-prefixed headers, plus **CONTENT\_LENGTH** and **CONTENT\_TYPE**.

## HttpRequest.headers

A case insensitive, dict-like object that provides access to all HTTP-prefixed headers (plus **Content-Length** and **Content-Type**) from the request.

The name of each header is stylized with title-casing (e.g. **User-Agent**) when it's displayed. You can access headers case-insensitively:

```
>>> request.headers
{'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6', ...}

>>> 'User-Agent' in request.headers
True
```



```
>>> 'user-agent' in request.headers
True

>>> request.headers['User-Agent']
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)

>>> request.headers['user-agent']
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)

>>> request.headers.get('User-Agent')
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)

>>> request.headers.get('user-agent')
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)
```

For use in, for example, Django templates, headers can also be looked up using underscores in place of hyphens:

```
{{ request.headers.user_agent }}
```

### **HttpRequest.resolver\_match**

An instance of **ResolverMatch** representing the resolved URL. This attribute is only set after URL resolving took place, which means it's available in all views but not in middleware which are executed before URL resolving takes place (you can use it in **process\_view()** though).

### Attributes set by application code

Django doesn't set these attributes itself but makes use of them if set by your application.

### **HttpRequest.current\_app**

The **url** template tag will use its value as the **current\_app** argument to **reverse()**.

### **HttpRequest.urlconf**

This will be used as the root URLconf for the current request, overriding the **ROOT\_URLCONF** setting. See [How Django processes a request](#) for details.

**urlconf** can be set to **None** to revert any changes made by previous middleware and return to using the **ROOT\_URLCONF**.

### **HttpRequest.exception\_reporter\_filter**

This will be used instead of **DEFAULT\_EXCEPTION\_REPORTER\_FILTER** for the current request. See [Custom error reports](#) for details.

### **HttpRequest.exception\_reporter\_class**

This will be used instead of **DEFAULT\_EXCEPTION\_REPORTER** for the current request. See [Custom error reports](#) for details.

## **Attributes set by middleware**

Some of the middleware included in Django's contrib apps set attributes on the request. If you don't see the attribute on a request, be sure the appropriate middleware class is listed in **MIDDLEWARE**.

### **HttpRequest.session**

From the **SessionMiddleware**: A readable and writable, dictionary-like object that represents the current session.

### **HttpRequest.site**

From the **CurrentSiteMiddleware**: An instance of **Site** or **RequestSite** as returned by **get\_current\_site()** representing the current site.

## HttpRequest.user

From the **AuthenticationMiddleware**: An instance of **AUTH\_USER\_MODEL** representing the currently logged-in user. If the user isn't currently logged in, **user** will be set to an instance of **AnonymousUser**. You can tell them apart with **is\_authenticated**, like so:

```
if request.user.is_authenticated:

    ... # Do something for logged-in users.

else:

    ... # Do something for anonymous users.
```

## Methods

### HttpRequest.get\_host()

Returns the originating host of the request using information from the **HTTP\_X\_FORWARDED\_HOST** (if **USE\_X\_FORWARDED\_HOST** is enabled) and **HTTP\_HOST** headers, in that order. If they don't provide a value, the method uses a combination of **SERVER\_NAME** and **SERVER\_PORT** as detailed in **PEP 3333**.

Example: "127.0.0.1:8000"

Raises **django.core.exceptions.DisallowedHost** if the host is not in **ALLOWED\_HOSTS** or the domain name is invalid according to **RFC 1034/1035**.

### Note

The **get\_host()** method fails when the host is behind multiple proxies. One solution is to use middleware to rewrite the proxy headers, as in the following example:

---

```
class MultipleProxyMiddleware:
```

```
    FORWARDED_FOR_FIELDS = [  
  
        'HTTP_X_FORWARDED_FOR',  
  
        'HTTP_X_FORWARDED_HOST',  
  
        'HTTP_X_FORWARDED_SERVER',  
  
    ]
```

```
def __init__(self, get_response):  
  
    self.get_response = get_response
```

```
def __call__(self, request):  
  
    """    Rewrites the proxy headers so that only the most  
  
    recent proxy is used.  
  
    """
```

```
for field in self.FORWARDED_FOR_FIELDS:
```

```
    if field in request.META:
```

```
        if ',' in request.META[field]:
```

```
            parts = request.META[field].split(',')
```

```
            request.META[field] = parts[-1].strip()
```

```
return self.get_response(request)
```

This middleware should be positioned before any other middleware that relies on the value of `get_host()` – for instance, `CommonMiddleware` or `CsrfViewMiddleware`.

### `HttpRequest.get_port()`

Returns the originating port of the request using information from the `HTTP_X_FORWARDED_PORT` (if `USE_X_FORWARDED_PORT` is enabled) and `SERVER_PORT META` variables, in that order.

### `HttpRequest.get_full_path()`

Returns the **path**, plus an appended query string, if applicable.

Example: `"/music/bands/the_beatles/?print=true"`

### `HttpRequest.get_full_path_info()`

Like `get_full_path()`, but uses **path\_info** instead of **path**.

Example: `"/minfo/music/bands/the_beatles/?print=true"`

### `HttpRequest.build_absolute_uri(location=None)`

Returns the absolute URI form of **location**. If no location is provided, the location will be set to `request.get_full_path()`.

If the location is already an absolute URI, it will not be altered. Otherwise the absolute URI is built using the server variables available in this request. For example:

```
>>> request.build_absolute_uri()

'https://example.com/music/bands/the_beatles/?print=true'

>>> request.build_absolute_uri('/bands/')
```

```
'https://example.com/bands/'

>>> request.build_absolute_uri('https://example2.com/bands/')

'https://example2.com/bands/'
```

### Note

Mixing HTTP and HTTPS on the same site is discouraged, therefore **build\_absolute\_uri()** will always generate an absolute URI with the same scheme the current request has. If you need to redirect users to HTTPS, it's best to let your web server redirect all HTTP traffic to HTTPS.

### **HttpRequest.get\_signed\_cookie(key, default=RAISE\_ERROR, salt="", max\_age=None)**

Returns a cookie value for a signed cookie, or raises a **django.core.signing.BadSignature** exception if the signature is no longer valid. If you provide the **default** argument the exception will be suppressed and that default value will be returned instead.

The optional **salt** argument can be used to provide extra protection against brute force attacks on your secret key. If supplied, the **max\_age** argument will be checked against the signed timestamp attached to the cookie value to ensure the cookie is not older than **max\_age** seconds.

For example:

```
>>> request.get_signed_cookie('name')

'Tony'

>>> request.get_signed_cookie('name', salt='name-salt')

'Tony' # assuming cookie was set using the same salt

>>> request.get_signed_cookie('nonexistent-cookie')

...
```

```
KeyError: 'nonexistent-cookie'

>>> request.get_signed_cookie('nonexistent-cookie', False)

False

>>> request.get_signed_cookie('cookie-that-was-tampered-with')

...

BadSignature: ...

>>> request.get_signed_cookie('name', max_age=60)

...

SignatureExpired: Signature age 1677.3839159 > 60 seconds

>>> request.get_signed_cookie('name', False, max_age=60)

False
```

See cryptographic signing for more information.

### **HttpRequest.is\_secure()**

Returns **True** if the request is secure; that is, if it was made with HTTPS.

### **HttpRequest.accepts(*mime\_type*)**

Returns **True** if the request **Accept** header matches the **mime\_type** argument:

```
>>> request.accepts('text/html')

True
```

Most browsers send **Accept: \*/\*** by default, so this would return **True** for all content types. Setting an explicit **Accept** header in API requests can be useful for returning a different content type for

— those consumers only. See Content negotiation example of using **accepts()** to return different content to API consumers.

If a response varies depending on the content of the **Accept** header and you are using some form of caching like Django's **cache middleware**, you should decorate the view with **vary\_on\_headers('Accept')** so that the responses are properly cached.

**HttpRequest.read(size=None)**

**HttpRequest.readline()**

**HttpRequest.readlines()**

**HttpRequest.\_\_iter\_\_()**

Methods implementing a file-like interface for reading from an **HttpRequest** instance. This makes it possible to consume an incoming request in a streaming fashion. A common use-case would be to process a big XML payload with an iterative parser without constructing a whole XML tree in memory.

Given this standard interface, an **HttpRequest** instance can be passed directly to an XML parser such as **ElementTree**:

```
import xml.etree.ElementTree as ET

for element in ET.iterparse(request):

    process(element)
```

**HttpResponse objects-**

**class HttpResponse**

In contrast to **HttpRequest** objects, which are created automatically by Django, **HttpResponse** objects are your responsibility. Each view you write is responsible for instantiating, populating, and returning an **HttpResponse**.

The **HttpResponse** class lives in the **django.http** module.



---

## Usage

### Passing strings

Typical usage is to pass the contents of the page, as a string, bytestring, or **memoryview**, to the **HttpResponse** constructor:

```
>>> from django.http import HttpResponse

>>> response = HttpResponse("Here's the text of the web page.")

>>> response = HttpResponse("Text only, please.", content_type="text/plain")

>>> response = HttpResponse(b'Bytestrings are also accepted.')

>>> response = HttpResponse(memoryview(b'Memoryview as well.'))
```

But if you want to add content incrementally, you can use **response** as a file-like object:

```
>>> response = HttpResponse()

>>> response.write("<p>Here's the text of the web page.</p>")

>>> response.write("<p>Here's another paragraph.</p>")
```

### Passing iterators

Finally, you can pass **HttpResponse** an iterator rather than strings. **HttpResponse** will consume the iterator immediately, store its content as a string, and discard it. Objects with a **close()** method such as files and generators are immediately closed.

If you need the response to be streamed from the iterator to the client, you must use the **StreamingHttpResponse** class instead.

## Setting header fields

To set or remove a header field in your response, use **HttpResponse.headers**:

```
>>> response = HttpResponse()
>>> response.headers['Age'] = 120
>>> del response.headers['Age']
```

You can also manipulate headers by treating your response like a dictionary:

```
>>> response = HttpResponse()
>>> response['Age'] = 120
>>> del response['Age']
```

This proxies to **HttpResponse.headers**, and is the original interface offered by **HttpResponse**.

When using this interface, unlike a dictionary, **del** doesn't raise **KeyError** if the header field doesn't exist.

You can also set headers on instantiation:

```
>>> response = HttpResponse(headers={'Age': 120})
```

For setting the **Cache-Control** and **Vary** header fields, it is recommended to use the **patch\_cache\_control()** and **patch\_vary\_headers()** methods from **django.utils.cache**, since these fields can have multiple, comma-separated values. The “patch” methods ensure that other values, e.g. added by a middleware, are not removed.

HTTP header fields cannot contain newlines. An attempt to set a header field containing a newline character (CR or LF) will raise **BadHeaderError**

### Changed in Django 3.2:

The **HttpResponse.headers** interface was added.

The ability to set headers on instantiation was added.

Telling the browser to treat the response as a file attachment¶

To tell the browser to treat the response as a file attachment, set the **Content-Type** and **Content-Disposition** headers. For example, this is how you might return a Microsoft Excel spreadsheet:

```
>>> response = HttpResponse(my_data, headers={
...     'Content-Type': 'application/vnd.ms-excel',
...     'Content-Disposition': 'attachment; filename="foo.xls"',
... })
```

There's nothing Django-specific about the **Content-Disposition** header, but it's easy to forget the syntax, so we've included it here.

## Attributes

### HttpResponse.content

A bytestring representing the content, encoded from a string if necessary.

### HttpResponse.headers

#### New in Django 3.2.

A case insensitive, dict-like object that provides an interface to all HTTP headers on the response. See [Setting header fields](#).

## HttpResponse.charset

A string denoting the charset in which the response will be encoded. If not given at **HttpResponse** instantiation time, it will be extracted from **content\_type** and if that is unsuccessful, the **DEFAULT\_CHARSET** setting will be used.

## HttpResponse.status\_code

The **HTTP status code** for the response.

Unless **reason\_phrase** is explicitly set, modifying the value of **status\_code** outside the constructor will also modify the value of **reason\_phrase**.

## HttpResponse.reason\_phrase

The HTTP reason phrase for the response. It uses the **HTTP standard's** default reason phrases.

Unless explicitly set, **reason\_phrase** is determined by the value of **status\_code**.

## HttpResponse.streaming

This is always **False**.

This attribute exists so middleware can treat streaming responses differently from regular responses.

## HttpResponse.closed

**True** if the response has been closed.

## Methods

**HttpResponse.\_\_init\_\_(content=b'', content\_type=None, status=200, reason=None, charset=None, headers=None)**

Instantiates an **HttpResponse** object with the given page content, content type, and headers.

**content** is most commonly an iterator, bytestring, **memoryview**, or string. Other types will be converted to a bytestring by encoding their string representation. Iterators should return strings or bytestrings and those will be joined together to form the content of the response.

**content\_type** is the MIME type optionally completed by a character set encoding and is used to fill the HTTP **Content-Type** header. If not specified, it is formed by **'text/html'** and the **DEFAULT\_CHARSET** settings, by default: **"text/html; charset=utf-8"**.

**status** is the **HTTP status code** for the response. You can use Python's **http.HTTPStatus** for meaningful aliases, such as **HTTPStatus.NO\_CONTENT**.

**reason** is the HTTP response phrase. If not provided, a default phrase will be used.

**charset** is the charset in which the response will be encoded. If not given it will be extracted from **content\_type**, and if that is unsuccessful, the **DEFAULT\_CHARSET** setting will be used.

**headers** is a **dict** of HTTP headers for the response.

### Changed in Django 3.2:

The **headers** parameter was added.

#### **HttpResponse.\_\_setitem\_\_(*header, value*)**

Sets the given header name to the given value. Both **header** and **value** should be strings.

#### **HttpResponse.\_\_delitem\_\_(*header*)**

Deletes the header with the given name. Fails silently if the header doesn't exist. Case-insensitive.

#### **HttpResponse.\_\_getitem\_\_(*header*)**

Returns the value for the given header name. Case-insensitive.

#### **HttpResponse.get(*header, alternate=None*)**

— Returns the value for the given header, or an **alternate** if the header doesn't exist.

### **HttpResponse.has\_header(*header*)**

Returns **True** or **False** based on a case-insensitive check for a header with the given name.

### **HttpResponse.items()**

Acts like **dict.items()** for HTTP headers on the response.

### **HttpResponse.setdefault(*header*, *value*)**

Sets a header unless it has already been set.

### **HttpResponse.set\_cookie(*key*, *value*=", *max\_age*=None, *expires*=None, *path*="/, *domain*=None, *secure*=False, *httponly*=False, *samesite*=None)**

Sets a cookie. The parameters are the same as in the **Morsel** cookie object in the Python standard library.

- **max\_age** should be an integer number of seconds, or **None** (default) if the cookie should last only as long as the client's browser session. If **expires** is not specified, it will be calculated.
- **expires** should either be a string in the format "**Wdy, DD-Mon-YY HH:MM:SS GMT**" or a **datetime.datetime** object in UTC. If **expires** is a **datetime** object, the **max\_age** will be calculated.
- Use **domain** if you want to set a cross-domain cookie. For example, **domain="example.com"** will set a cookie that is readable by the domains

---

www.example.com, blog.example.com, etc. Otherwise, a cookie will only be readable by the domain that set it.

- Use **secure=True** if you want the cookie to be only sent to the server when a request is made with the **https** scheme.
- Use **httponly=True** if you want to prevent client-side JavaScript from having access to the cookie.

HttpOnly is a flag included in a Set-Cookie HTTP response header. It's part of the **RFC 6265** standard for cookies and can be a useful way to mitigate the risk of a client-side script accessing the protected cookie data.

- Use **samesite='Strict'** or **samesite='Lax'** to tell the browser not to send this cookie when performing a cross-origin request. SameSite isn't supported by all browsers, so it's not a replacement for Django's CSRF protection, but rather a defense in depth measure.

Use **samesite='None'** (string) to explicitly state that this cookie is sent with all same-site and cross-site requests.

### Warning

**RFC 6265** states that user agents should support cookies of at least 4096 bytes. For many browsers this is also the maximum size. Django will not raise an exception if there's an attempt to store a cookie of more than 4096 bytes, but many browsers will not set the cookie correctly.

**HttpResponse.set\_signed\_cookie(key, value, salt="", max\_age=None, expires=None, path='/', domain=None, secure=False, httponly=False, samesite=None)**

Like **set\_cookie()**, but cryptographic signing the cookie before setting it. Use in conjunction with **HttpRequest.get\_signed\_cookie()**. You can use the optional **salt** argument for added key

— strength, but you will need to remember to pass it to the corresponding **HttpRequest.get\_signed\_cookie()** call.

### **HttpResponse.delete\_cookie(key, path='/', domain=None, samesite=None)**

Deletes the cookie with the given key. Fails silently if the key doesn't exist.

Due to the way cookies work, **path** and **domain** should be the same values you used in **set\_cookie()** – otherwise the cookie may not be deleted.

### **HttpResponse.close()**

This method is called at the end of the request directly by the WSGI server.

### **HttpResponse.write(content)**

This method makes an **HttpResponse** instance a file-like object.

### **HttpResponse.flush()**

This method makes an **HttpResponse** instance a file-like object.

### **HttpResponse.tell()**

This method makes an **HttpResponse** instance a file-like object.

### **HttpResponse.getvalue()**

Returns the value of **HttpResponse.content**. This method makes an **HttpResponse** instance a stream-like object.

### **HttpResponse.readable()**

Always **False**. This method makes an **HttpResponse** instance a stream-like object.

### **HttpResponse.seekable()**



— Always **False**. This method makes an **HttpResponse** instance a stream-like object.

### **HttpResponse.writable()**

Always **True**. This method makes an **HttpResponse** instance a stream-like object.

### **HttpResponse.writelines(*lines*)**

Writes a list of lines to the response. Line separators are not added. This method makes an **HttpResponse** instance a stream-like object.

## **HttpResponse subclasses**

Django includes a number of **HttpResponse** subclasses that handle different types of HTTP responses. Like **HttpResponse**, these subclasses live in **django.http**.

### **class HttpResponseRedirect**

The first argument to the constructor is required – the path to redirect to. This can be a fully qualified URL (e.g. **'https://www.yahoo.com/search/'**), an absolute path with no domain (e.g. **'/search/'**), or even a relative path (e.g. **'search/'**). In that last case, the client browser will reconstruct the full URL itself according to the current path. See **HttpResponse** for other optional constructor arguments. Note that this returns an HTTP status code 302.

#### **url**

This read-only attribute represents the URL the response will redirect to (equivalent to the **Location** response header).

### **class HttpResponseRedirect**

Like **HttpResponseRedirect**, but it returns a permanent redirect (HTTP status code 301) instead of a “found” redirect (status code 302).

### **`class HttpResponseNotModified`**

The constructor doesn't take any arguments and no content should be added to this response. Use this to designate that a page hasn't been modified since the user's last request (status code 304).

### **`class HttpResponseBadRequest`**

Acts just like **`HttpResponse`** but uses a 400 status code.

### **`class HttpResponseNotFound`**

Acts just like **`HttpResponse`** but uses a 404 status code.

### **`class HttpResponseForbidden`**

Acts just like **`HttpResponse`** but uses a 403 status code.

### **`class HttpResponseNotAllowed`**

Like **`HttpResponse`**, but uses a 405 status code. The first argument to the constructor is required: a list of permitted methods (e.g. **`['GET', 'POST']`**).

### **`class HttpResponseGone`**

Acts just like **`HttpResponse`** but uses a 410 status code.

### **`class HttpResponseServerError`**

Acts just like **`HttpResponse`** but uses a 500 status code.

### **Note**

If a custom subclass of **HttpResponse** implements a **render** method, Django will treat it as emulating a **SimpleTemplateResponse**, and the **render** method must itself return a valid response object.

### Custom response classes

If you find yourself needing a response class that Django doesn't provide, you can create it with the help of **http.HTTPStatus**. For example:

```
from http import HTTPStatus

from django.http import HttpResponse

class HttpResponseNoContent(HttpResponse):

    status_code = HTTPStatus.NO_CONTENT
```

### JsonResponse objects

**class JsonResponse(data, encoder=DjangoJSONEncoder, safe=True, json\_dumps\_params=None, \*\*kwargs)**

An **HttpResponse** subclass that helps to create a JSON-encoded response. It inherits most behavior from its superclass with a couple differences:

Its default **Content-Type** header is set to *application/json*.

The first parameter, **data**, should be a **dict** instance. If the **safe** parameter is set to **False** (see below) it can be any JSON-serializable object.

The **encoder**, which defaults to **django.core.serializers.json.DjangoJSONEncoder**, will be used to serialize the data. See JSON serialization for more details about this serializer.

The **safe** boolean parameter defaults to **True**. If it's set to **False**, any object can be passed for serialization (otherwise only **dict** instances are allowed). If **safe** is **True** and a non-**dict** object is passed as the first argument, a **TypeError** will be raised.

The **json\_dumps\_params** parameter is a dictionary of keyword arguments to pass to the **json.dumps()** call used to generate the response.

## Usage

Typical usage could look like:

```
>>> from django.http import JsonResponse
>>> response = JsonResponse({'foo': 'bar'})
>>> response.content
b'{"foo": "bar"}'
```

### Serializing non-dictionary objects

In order to serialize objects other than **dict** you must set the **safe** parameter to **False**:

```
>>> response = JsonResponse([1, 2, 3], safe=False)
```

Without passing **safe=False**, a **TypeError** will be raised.

Note that an API based on **dict** objects is more extensible, flexible, and makes it easier to maintain forwards compatibility. Therefore, you should avoid using non-dict objects in JSON-encoded response.

### Warning

Before the 5th edition of ECMAScript it was possible to poison the JavaScript **Array** constructor. For this reason, Django does not allow passing non-dict objects to the **JsonResponse** constructor by default. However, most modern browsers implement ECMAScript 5 which removes this attack vector. Therefore it is possible to disable this security precaution.

Changing the default JSON encoder

If you need to use a different JSON encoder class you can pass the **encoder** parameter to the constructor method:

```
>>> response = JsonResponse(data, encoder=MyJSONEncoder)
```

## Working with forms

### About this document

This document provides an introduction to the basics of web forms and how they are handled in Django. For a more detailed look at specific areas of the forms API, see The Forms API, Form fields, and Form and field validation.

Unless you're planning to build websites and applications that do nothing but publish content, and don't accept input from your visitors, you're going to need to understand and use forms.

Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

## 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 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.

### Django's role in forms

Handling forms is a complex business. Consider Django's admin, where numerous items of data of several different types may need to be prepared for display in a form, rendered as HTML, edited using a

convenient interface, returned to the server, validated and cleaned up, and then saved or passed on for further processing.

Django's form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client

It is *possible* to write code that does all of this manually, but Django can take care of it all for you.

## Forms in Django

We've described HTML forms briefly, but an HTML **<form>** is just one part of the machinery required.

In the context of a web application, 'form' might refer to that HTML **<form>**, or to the Django **Form** that produces it, or to the structured data returned when it is submitted, or to the end-to-end working collection of these parts.

### The Django Form class

At the heart of this system of components is Django's **Form** class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a **Form** class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form **<input>** elements. (A **ModelForm** maps a model class's fields to HTML form **<input>** elements via a **Form**; this is what the Django admin is based upon.)



A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A **DateField** and a **FileField** handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML “widget” - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

## Instantiating, processing, and rendering forms

When rendering an object in Django, we generally:

1. get hold of it in the view (fetch it from the database, for example)
2. pass it to the template context
3. expand it to HTML markup using template variables

Rendering a form in a template involves nearly the same work as rendering any other kind of object, but there are some key differences.

In the case of a model instance that contained no data, it would rarely if ever be useful to do anything with it in a template. On the other hand, it makes perfect sense to render an unpopulated form - that's what we do when we want the user to populate it.

So when we handle a model instance in a view, we typically retrieve it from the database. When we're dealing with a form we typically instantiate it in the view.

When we instantiate a form, we can opt to leave it empty or pre-populate it, for example with:

- data from a saved model instance (as in the case of admin forms for editing)
- data that we have collated from other sources
- data received from a previous HTML form submission

The last of these cases is the most interesting, because it's what makes it possible for users not just to read a website, but to send information back to it too.

Building a form

## The work that needs to be done

Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```
<form action="/your-name/" method="post">

  <label for="your_name">Your name: </label>

  <input id="your_name" type="text" name="your_name" value="{{ current_name }}">

  <input type="submit" value="OK">

</form>
```

This tells the browser to return the form data to the URL **/your-name/**, using the **POST** method. It will display a text field, labeled “Your name:”, and a button marked “OK”. If the template context contains a **current\_name** variable, that will be used to pre-fill the **your\_name** field.

You'll need a view that renders the template containing the HTML form, and that can supply the **current\_name** field as appropriate.

When the form is submitted, the **POST** request which is sent to the server will contain the form data.

Now you'll also need a view corresponding to that **/your-name/** URL which will find the appropriate key/value pairs in the request, and then process them.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be pre-populated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it's much easier to get Django to do most of this work for us.

## Building a form in Django

### 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:

```
<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.

## Forms and Cross Site Request Forgery protection

Django ships with an easy-to-use protection against Cross Site Request Forgeries. When submitting a form via **POST** with CSRF protection enabled you must use the **`csrf_token`** template tag as in the preceding example. However, since CSRF protection is not directly tied to forms in templates, this tag is omitted from the following examples in this document.

## HTML5 input types and browser validation

If your form includes a **`URLField`**, an **`EmailField`** or any integer field type, Django will use the **`url`**, **`email`** and **`number`** HTML5 input types. By default, browsers may apply their own validation on these fields, which may be stricter than Django's validation. If you would like to disable this behavior, set the **`novalidate`** attribute on the **`form`** tag, or specify a different widget on the field, like **`TextInput`**.



We now have a working web form, described by a Django **Form**, processed by a view, and rendered as an HTML **<form>**.

That's all you need to get started, but the forms framework puts a lot more at your fingertips. Once you understand the basics of the process described above, you should be prepared to understand other features of the forms system and ready to learn a bit more about the underlying machinery.

## More about Django Form classes

All form classes are created as subclasses of either **django.forms.Form** or **django.forms.ModelForm**. You can think of **ModelForm** as a subclass of **Form**. **Form** and **ModelForm** actually inherit common functionality from a (private) **BaseForm** class, but this implementation detail is rarely important.

## Models and Forms

In fact if your form is going to be used to directly add or edit a Django model, a **ModelForm** can save you a great deal of time, effort, and code, because it will build a form, along with the appropriate fields and their attributes, from a **Model** class.

## Bound and unbound form instances

The distinction between Bound and unbound forms is important:

- An unbound form has no data associated with it. When rendered to the user, it will be empty or will contain default values.
- A bound form has submitted data, and hence can be used to tell if that data is valid. If an invalid bound form is rendered, it can include inline error messages telling the user what data to correct.

The form's **is\_bound** attribute will tell you whether a form has data bound to it or not.

## More on fields

Consider a more useful form than our minimal example above, which we could use to implement “contact me” functionality on a personal website:

forms.py

```
from django import forms

class ContactForm(forms.Form):

    subject = forms.CharField(max_length=100)

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

    sender = forms.EmailField()

    cc_myself = forms.BooleanField(required=False)
```

Our earlier form used a single field, **your\_name**, a **CharField**. In this case, our form has four fields: **subject**, **message**, **sender** and **cc\_myself**. **CharField**, **EmailField** and **BooleanField** are just three of the available field types; a full list can be found in Form fields.

## 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.

## Field data

Whatever the data submitted with a form, once it has been successfully validated by calling **is\_valid()** (and **is\_valid()** has returned **True**), the validated form data will be in the **form.cleaned\_data** dictionary. This data will have been nicely converted into Python types for you.

## Note



You can still access the unvalidated data directly from **request.POST** at this point, but the validated data is better.

In the contact form example above, **cc\_myself** will be a boolean value. Likewise, fields such as **IntegerField** and **FloatField** convert values to a Python **int** and **float** respectively.

Here's how the form data could be processed in the view that handles this form:

views.py

```
from django.core.mail import send_mail

if form.is_valid():

    subject = form.cleaned_data['subject']

    message = form.cleaned_data['message']

    sender = form.cleaned_data['sender']

    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']

    if cc_myself:

        recipients.append(sender)

    send_mail(subject, message, sender, recipients)

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

## Tip

For more on sending email from Django, see [Sending email](#).

Some field types need some extra handling. For example, files that are uploaded using a form need to be handled differently (they can be retrieved from **request.FILES**, rather than **request.POST**). For details of how to handle file uploads with your form, see [Binding uploaded files to a form](#).

## 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.

## Form rendering options

### Additional form template furniture

Don't forget that a form's output does *not* include the surrounding **<form>** tags, or the form's **submit** control. You will have to provide these yourself.

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>
```

Note that each form field has an ID attribute set to **id\_<field-name>**, which is referenced by the accompanying label tag. This is important in ensuring that forms are accessible to assistive technology such as screen reader software. You can also customize the way in which labels and ids are generated.

See [Outputting forms as HTML](#) for more on this.

## 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>
```

Complete **<label>** elements can also be generated using the **label\_tag()**. For example:

```
<div class="fieldWrapper">

    {{ form.subject.errors }}

    {{ form.subject.label_tag }}

    {{ form.subject }}
```

```
</div>
```

## Rendering form error messages

The price of this flexibility is a bit more work. Until now we haven't had to worry about how to display form errors, because that's taken care of for us. In this example we have had to make sure we take care of any errors for each field and any errors for the form as a whole. Note `{{ form.non_field_errors }}` at the top of the form and the template lookup for errors on each field.

Using `{{ form.name_of_field.errors }}` displays a list of form errors, rendered as an unordered list. This might look like:

```
<ul class="errorlist">

    <li>Sender is required.</li>

</ul>
```

The list has a CSS class of **errorlist** to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:

```
{% if form.subject.errors %}

    <ol>

        {% for error in form.subject.errors %}

            <li><strong>{{ error|escape }}</strong></li>

        {% endfor %}

    </ol>

{% endif %}
```

Non-field errors (and/or hidden field errors that are rendered at the top of the form when using helpers like **form.as\_p()**) will be rendered with an additional class of **nonfield** to help distinguish them from field-specific errors. For example, **{{ form.non\_field\_errors }}** would look like:

```
<ul class="errorlist nonfield">

  <li>Generic validation error</li>

</ul>
```

See The Forms API for more on errors, styling, and working with form attributes in templates.

## Looping over the form's fields

If you're using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a **{% for %}** loop:

```
{% for field in form %}

  <div class="fieldWrapper">

    {{ field.errors }}

    {{ field.label_tag }} {{ field }}

    {% if field.help_text %}

      <p class="help">{{ field.help_text|safe }}</p>

    {% endif %}

  </div>

{% endfor %}
```

Useful attributes on **{{ field }}** include:

### **{{ field.label }}**

The label of the field, e.g. **Email address**.

### **{{ field.label\_tag }}**

The field's label wrapped in the appropriate HTML **<label>** tag. This includes the form's **label\_suffix**. For example, the default **label\_suffix** is a colon:

```
<label for="id_email">Email address:</label>
```

### **{{ field.id\_for\_label }}**

The ID that will be used for this field (**id\_email** in the example above). If you are constructing the label manually, you may want to use this in lieu of **label\_tag**. It's also useful, for example, if you have some inline JavaScript and want to avoid hardcoding the field's ID.

### **{{ field.value }}**

The value of the field. e.g **someone@example.com**.

### **{{ field.html\_name }}**

The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set.

### **{{ field.help\_text }}**

Any help text that has been associated with the field.

### **{{ field.errors }}**

Outputs a **<ul class="errorlist">** containing any validation errors corresponding to this field. You can customize the presentation of the errors with a **{% for error in field.errors %}** loop. In this case, each object in the loop is a string containing the error message.

### **{{ field.is\_hidden }}**

This attribute is **True** if the form field is a hidden field and **False** otherwise. It's not particularly

useful as a template variable, but could be useful in conditional tests such as:

```
{% if field.is_hidden %}

    {# Do something special #}

{% endif %}
```

### **{{ field.field }}**

The **Field** instance from the form class that this **BoundField** wraps. You can use it to access **Field** attributes, e.g. **{{ char\_field.field.max\_length }}**.

### **Templates-**

- Templates are the third and most important part of Django's MVT Structure. A template in Django is basically written in HTML, CSS, and Javascript in a .html file. Django framework efficiently handles and generates dynamically HTML web pages that are visible to the end-user.
- Django mainly functions with a backend so, in order to provide a frontend and provide a layout to our website, we use templates.
- There are two methods of adding the template to our website depending on our needs.
  - 1) We can use a single template directory which will be spread over the entire project.
  - 2) For each app of our project, we can create a different template directory.

### **The Django template language-**

- A Django template is 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.
- The main characteristics of Django Template language are Variables, Tags, Filters, and Comments.



## Main characteristics of Django templates –

### 1) Variables

Variables output a value from the context, which is a dict-like object mapping keys to values. The context object we sent from the view can be accessed in the template using variables of Django Template.

#### Syntax

```
{{ variable_name }}
```

#### Example

-Variables are surrounded by {{ and }} like this:

**Eg.-**My first name is {{ first\_name }}. My last name is {{ last\_name }}.

With a context of {'first\_name': 'Pawan', 'last\_name': 'Kumar'}, this template renders to:

**My first name is Pawan. My last name is Kumar.**

### 2) Tags –

It provide arbitrary logic in the rendering process

#### Syntax-

```
{% tag_name %}
```

#### Example-

Tags are surrounded by {% and %} like this:

```
{% csrf_token %}
```

Most tags accept arguments, for example :

```
{% cycle 'odd' 'even' %}
```

### 3) Filters

Django Template Engine provides filters that are used to transform the values of variables and tag

arguments.

Tags can't modify the value of a variable whereas filters can be used for incrementing the value of a variable or modifying it to one's own need.

**Syntax-**

```
{{ variable_name | filter_name }}
```

Filters can be “chained.” The output of one filter is applied to the next.

`{{ text|escape|linebreaks }}` is a common idiom for escaping text contents, then converting line breaks to `<p>` tags.

**Example-**

```
{{ value | length }}
```

If value is ['a', 'b', 'c', 'd'], the output will be 4.

**4) Comments**

Template 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.

**Syntax-**

```
{% comment 'comment_name' %}
```

```
{% endcomment %}
```

**Example-**

```
{% comment "Optional note" %}
```

```
Commented out text with {{ create_date|date:"c" }}
```

```
{% endcomment %}
```

**5) Template Inheritance**

Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines blocks that child templates can override. `extends` tag is used for the inheritance of templates in Django. One needs to repeat the same code again and again. Using `extends` we can inherit templates as well as variables.

## Syntax

```
{% extends 'template_name.html' %}
```

## Example :

assume the following directory structure:

```
dir1/  
    template.html  
    base2.html  
    my/  
        base3.html  
base1.html
```

In template.html, the following paths would be valid:

```
{% extends "../base2.html" %}  
{% extends "../base1.html" %}  
{% extends "../my/base3.html" %}
```

## Example on Django Template : Let us create one template and render it

### Step1) Create view.py

```
# import Http Response from django  
from django.shortcuts import render  
  
# create a function  
def learn_view(request):  
    # create a dictionary to pass  
    # data to the template
```

---

```
context = {
    "data": "Updating from the list",
    "list": ['Data Science', 'Python', 'Django', 'HTML5', 'JavaScript']
}
# return response with template and context
return render(request, "learn.html", context)
```

### Step2) URL Mapping: open urls.py

```
from django.urls import path

# importing views from views..py
from .views import learn_view

urlpatterns = [
    path("", learn_view),
]
```

### Step3) Create template

Create folder named as template and create new file names learn.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Homepage</title>
```

```
</head>
<body>
  <h1>Welcome to Learn Django.</h1>
  <p> Data is {{ data }}</p>
  <h4>List is </h4>
  <ul>
    {% for i in list %}
    <li>{{ i }}</li>
    {% endfor %}
  </ul>
</body>
</html>
```

#### **Step4) Open settings.py and Configure TEMPLATES section**

Copy path of your template directory and paste under TEMPLATES section in settings.py

```
'DIRS': ['C:\Users\Hp\Desktop\learndjango\learndjango\templates'],
```

#### **Step5) check output**

-Open browser and Visit <http://127.0.0.1:8000/>



## Django Models

- 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.
- A Django model is the built-in feature that Django uses to create tables, their fields, and various constraints. In short, Django Models is the SQL of Database one uses with Django.
- SQL (Structured Query Language) is complex and involves a lot of different queries for creating, deleting, updating or any other stuff related to database. Django models simplify the tasks and organize tables into models. Generally, each model maps to a single database table.
- Django models provide simplicity, consistency, version control and advanced metadata handling.
- Basics of a model include –
  - 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

## Quick Example : Creating Models

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 **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
);
```

### Fields types : -creating model Fields

- Each field in your model should be an instance of the appropriate Field class. Django uses the field class types to determine a few things:
  - The column type, which tells the database what kind of data to store (e.g. INTEGER, VARCHAR, TEXT).
  - The default HTML widget to use when rendering a form field (e.g. `<input type="text">`, `<select>`).
  - The minimal validation requirements, used in Django's admin and in automatically-generated forms.
- Django ships with dozens of built-in field types; you can find the complete list in the [model field reference](#).
- Few Filed types are listed below:

---

AutoField	IntegerField
CharField	TextField
DateField	FileField
ImageField	EmailField
DecimalField	JSONField

### Fields options : -creating model Fields

- Each field takes a certain set of field-specific arguments. For example, CharField (and its subclasses) require a max\_length argument which specifies the size of the VARCHAR database field used to store the data.
- Here's a quick summary of the most often-used ones:
  - 1) null -If True, Django will store empty values as NULL in the database. Default is False.
  - 2) blank -If True, the field is allowed to be blank. Default is False.
  - 3) choices- A sequence of 2-tuples to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES = [
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'), ]
```

## SQL operations in Django

### Making queries



Once you've created your data models, Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. This document explains how to use this API. Refer to the data model reference for full details of all the various model lookup options.

Throughout this guide (and in the reference), we'll refer to the following models, which comprise a Weblog application:

```
from django.db import models

class Blog(models.Model):

    name = models.CharField(max_length=100)

    tagline = models.TextField()

    def __str__(self):

        return self.name

class Author(models.Model):

    name = models.CharField(max_length=200)

    email = models.EmailField()

    def __str__(self):

        return self.name
```

```
class Entry(models.Model):

    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)

    headline = models.CharField(max_length=255)

    body_text = models.TextField()

    pub_date = models.DateField()

    mod_date = models.DateField()

    authors = models.ManyToManyField(Author)

    number_of_comments = models.IntegerField()

    number_of_pingbacks = models.IntegerField()

    rating = models.IntegerField()

    def __str__(self):

        return self.headline
```

## Creating objects

To represent database-table data in Python objects, Django uses an intuitive system: A model class represents a database table, and an instance of that class represents a particular record in the database table.

To create an object, instantiate it using keyword arguments to the model class, then call **save()** to save it to the database.

Assuming models live in a file **mysite/blog/models.py**, here's an example:

```
>>> from blog.models import Blog

>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')

>>> b.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.

### See also

**save()** takes a number of advanced options not described here. See the documentation for **save()** for complete details.

To create and save an object in a single step, use the **create()** method.

## Saving changes to objects

To save changes to an object that's already in the database, use **save()**.

Given a **Blog** instance **b5** that has already been saved to the database, this example changes its name and updates its record in the database:

```
>>> b5.name = 'New name'

>>> b5.save()
```

This performs an **UPDATE** SQL statement behind the scenes. Django doesn't hit the database until you explicitly call **save()**.

## Saving ForeignKey and ManyToManyField fields

Updating a **ForeignKey** field works exactly the same way as saving a normal field – assign an object of the right type to the field in question. This example updates the **blog** attribute of an **Entry** instance **entry**, assuming appropriate instances of **Entry** and **Blog** are already saved to the database (so we can retrieve them below):

```
>>> from blog.models import Blog, Entry

>>> entry = Entry.objects.get(pk=1)

>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")

>>> entry.blog = cheese_blog

>>> entry.save()
```

Updating a **ManyToManyField** works a little differently – use the **add()** method on the field to add a record to the relation. This example adds the **Author** instance **joe** to the **entry** object:

```
>>> from blog.models import Author

>>> joe = Author.objects.create(name="Joe")

>>> entry.authors.add(joe)
```

To add multiple records to a **ManyToManyField** in one go, include multiple arguments in the call to **add()**, like this:

```
>>> john = Author.objects.create(name="John")

>>> paul = Author.objects.create(name="Paul")

>>> george = Author.objects.create(name="George")

>>> ringo = Author.objects.create(name="Ringo")

>>> entry.authors.add(john, paul, george, ringo)
```

Django will complain if you try to assign or add an object of the wrong type.

## Retrieving 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. Access it directly via the model class, like so:

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>

>>> b = Blog(name='Foo', tagline='Bar')

>>> b.objects

Traceback:

...

AttributeError: "Manager isn't accessible via Blog instances."
```

### Note

**Managers** are accessible only via model classes, rather than from model instances, to enforce a separation between “table-level” operations and “record-level” operations.

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.

## Retrieving all objects

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**:

```
>>> all_entries = Entry.objects.all()
```

The **all()** method returns a **QuerySet** of all the objects in the database.

## Retrieving specific objects with filters-

The **QuerySet** returned by **all()** describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects.

To create such a subset, you refine the initial **QuerySet**, adding filter conditions. The two most common ways to refine a **QuerySet** are:

### **filter(\*\*kwargs)**

Returns a new **QuerySet** containing objects that match the given lookup parameters.

### **exclude(\*\*kwargs)**

Returns a new **QuerySet** containing objects that do *not* match the given lookup parameters.

The lookup parameters (**\*\*kwargs** in the above function definitions) should be in the format described in [Field lookups](#) below.

For example, to get a **QuerySet** of blog entries from the year 2006, use **filter()** like so:

```
Entry.objects.filter(pub_date__year=2006)
```

With the default manager class, it is the same as:

```
Entry.objects.all().filter(pub_date__year=2006)
```

## Chaining filters-

The result of refining a **QuerySet** is itself a **QuerySet**, so it's possible to chain refinements together. For example:

```
>>> Entry.objects.filter(  
...     headline__startswith='What'  
... ).exclude(  
...     pub_date__gte=datetime.date.today()  
... ).filter(  
...     pub_date__gte=datetime.date(2005, 1, 30)  
... )
```

This takes the initial **QuerySet** of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a **QuerySet** containing all entries with a headline that starts with “What”, that were published between January 30, 2005, and the current day.

Filtered QuerySets are unique

Each time you refine a **QuerySet**, you get a brand-new **QuerySet** that is in no way bound to the previous **QuerySet**. Each refinement creates a separate and distinct **QuerySet** that can be stored, used and reused.

Example:

```
>>> q1 = Entry.objects.filter(headline__startswith="What")  
  
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())  
  
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

These three **QuerySets** are separate. The first is a base **QuerySet** containing all entries that contain a headline starting with “What”. The second is a subset of the first, with an additional criteria that excludes records whose **pub\_date** is today or in the future. The third is a subset of the first, with an additional criteria that selects only the records whose **pub\_date** is today or in the future. The initial **QuerySet** (**q1**) is unaffected by the refinement process.

### QuerySets are lazy-

**QuerySets** are lazy – the act of creating a **QuerySet** doesn’t involve any database activity. You can stack filters together all day long, and Django won’t actually run the query until the **QuerySet** is *evaluated*. Take a look at this example:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

Though this looks like three database hits, in fact it hits the database only once, at the last line (**print(q)**). In general, the results of a **QuerySet** aren’t fetched from the database until you “ask” for them. When you do, the **QuerySet** is *evaluated* by accessing the database. For more details on exactly when evaluation takes place, see [When QuerySets are evaluated](#).

### Retrieving a single object with get()-

**filter()** will always give you a **QuerySet**, even if only a single object matches the query - in this case, it will be a **QuerySet** containing a single element.

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:



```
>>> one_entry = Entry.objects.get(pk=1)
```

You can use any query expression with **get()**, just like with **filter()** - again, see [Field lookups](#) below.

Note that there is a difference between using **get()**, and using **filter()** with a slice of **[0]**. If there are no results that match the query, **get()** will raise a **DoesNotExist** exception. This exception is an attribute of the model class that the query is being performed on - so in the code above, if there is no **Entry** object with a primary key of 1, Django will raise **Entry.DoesNotExist**.

Similarly, Django will complain if more than one item matches the **get()** query. In this case, it will raise **MultipleObjectsReturned**, which again is an attribute of the model class itself.

## Other QuerySet methods-

Most of the time you'll use **all()**, **get()**, **filter()** and **exclude()** when you need to look up objects from the database. However, that's far from all there is; see the QuerySet API Reference for a complete list of all the various **QuerySet** methods.

## Limiting QuerySets-

Use a subset of Python's array-slicing syntax to limit your **QuerySet** to a certain number of results. This is the equivalent of SQL's **LIMIT** and **OFFSET** clauses.

For example, this returns the first 5 objects (**LIMIT 5**):

```
>>> Entry.objects.all()[:5]
```

This returns the sixth through tenth objects (**OFFSET 5 LIMIT 5**):

```
>>> Entry.objects.all()[5:10]
```

Negative indexing (i.e. **Entry.objects.all()[-1]**) is not supported.

Generally, slicing a **QuerySet** returns a new **QuerySet** – it doesn't evaluate the query. An exception is if you use the “step” parameter of Python slice syntax. For example, this would actually execute the query in order to return a list of every *second* object of the first 10:

```
>>> Entry.objects.all()[:10:2]
```

Further filtering or ordering of a sliced queryset is prohibited due to the ambiguous nature of how that might work.

To retrieve a *single* object rather than a list (e.g. **SELECT foo FROM bar LIMIT 1**), use an index instead of a slice. For example, this returns the first **Entry** in the database, after ordering entries alphabetically by headline:

```
>>> Entry.objects.order_by('headline')[0]
```

This is roughly equivalent to:

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

Note, however, that the first of these will raise **IndexError** while the second will raise **DoesNotExist** if no objects match the given criteria. See **get()** for more details.

# Handling sessions

## What is session -

- The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID – not the data itself (unless you're using the cookie-based backend).
- Django provides full support for anonymous sessions.

## Enabling sessions

Sessions are implemented via a piece of middleware.

To enable session functionality, do the following:

- Edit the **MIDDLEWARE** setting and make sure it contains **'django.contrib.sessions.middleware.SessionMiddleware'**. The default **settings.py** created by **django-admin startproject** has **SessionMiddleware** activated.

If you don't want to use sessions, you might as well remove the **SessionMiddleware** line from **MIDDLEWARE** and **'django.contrib.sessions'** from your **INSTALLED\_APPS**. It'll save you a small bit of overhead.

## Configuring the session engine-

By default, Django stores sessions in your database (using the model **django.contrib.sessions.models.Session**). Though this is convenient, in some setups it's faster to store session data elsewhere, so Django can be configured to store session data on your filesystem or in your cache.

## Using database-backed sessions-

If you want to use a database-backed session, you need to add **'django.contrib.sessions'** to your **INSTALLED\_APPS** setting.

Once you have configured your installation, run **manage.py migrate** to install the single database table that stores session data.

## Using cached sessions-

For better performance, you may want to use a cache-based session backend.

To store session data using Django's cache system, you'll first need to make sure you've configured your cache; see the [cache documentation](#) for details.

### Warning

You should only use cache-based sessions if you're using the Memcached or Redis cache backend. The local-memory cache backend doesn't retain data long enough to be a good choice, and it'll be faster to use file or database sessions directly instead of sending everything through the file or database cache backends. Additionally, the local-memory cache backend is NOT multi-process safe, therefore probably not a good choice for production environments.

If you have multiple caches defined in **CACHES**, Django will use the default cache. To use another cache, set **SESSION\_CACHE\_ALIAS** to the name of that cache.

Once your cache is configured, you've got two choices for how to store data in the cache:

- Set **SESSION\_ENGINE** to **"django.contrib.sessions.backends.cache"** for a simple caching session store. Session data will be stored directly in your cache. However, session data may not be persistent: cached data can be evicted if the cache fills up or if the cache server is restarted.
- For persistent, cached data, set **SESSION\_ENGINE** to **"django.contrib.sessions.backends.cached\_db"**. This uses a write-through cache – every write to the cache will also be written to the database. Session reads only use the database if the data is not already in the cache.

Both session stores are quite fast, but the simple cache is faster because it disregards persistence. In most cases, the **cached\_db** backend will be fast enough, but if you need that last bit of performance, and are willing to let session data be expunged from time to time, the **cache** backend is for you.

If you use the **cached\_db** session backend, you also need to follow the configuration instructions for the [using database-backed sessions](#).

### Using file-based sessions-

To use file-based sessions, set the **SESSION\_ENGINE** setting to **"django.contrib.sessions.backends.file"**.

You might also want to set the **SESSION\_FILE\_PATH** setting (which defaults to output from **tempfile.gettempdir()**, most likely **/tmp**) to control where Django stores session files. Be sure to check that your web server has permissions to read and write to this location.

### Using cookie-based sessions-

To use cookies-based sessions, set the **SESSION\_ENGINE** setting to **"django.contrib.sessions.backends.signed\_cookies"**. The session data will be stored using Django's tools for cryptographic signing and the **SECRET\_KEY** setting.

#### Note

It's recommended to leave the **SESSION\_COOKIE\_HTTPONLY** setting on **True** to prevent access to the stored data from JavaScript.

#### Warning

If the **SECRET\_KEY** is not kept secret and you are using the **PickleSerializer**, this can lead to arbitrary remote code execution.

An attacker in possession of the **SECRET\_KEY** can not only generate falsified session data, which your site will trust, but also remotely execute arbitrary code, as the data is serialized using pickle.

If you use cookie-based sessions, pay extra care that your secret key is always kept completely secret, for any system which might be remotely accessible.

### **The session data is signed but not encrypted**

When using the cookies backend the session data can be read by the client.

A MAC (Message Authentication Code) is used to protect the data against changes by the client, so that the session data will be invalidated when being tampered with. The same invalidation happens if the client storing the cookie (e.g. your user's browser) can't store all of the session cookie and drops data. Even though Django compresses the data, it's still entirely possible to exceed the **common limit of 4096 bytes** per cookie.

### **No freshness guarantee**

Note also that while the MAC can guarantee the authenticity of the data (that it was generated by your site, and not someone else), and the integrity of the data (that it is all there and correct), it cannot guarantee freshness i.e. that you are being sent back the last thing you sent to the client. This means that for some uses of session data, the cookie backend might open you up to replay attacks. Unlike other session backends which keep a server-side record of each session and invalidate it when a user logs out, cookie-based sessions are not invalidated when a user logs out. Thus if an attacker steals a user's cookie, they can use that cookie to login as that user even if the user logs out. Cookies will only be detected as 'stale' if they are older than your **SESSION\_COOKIE\_AGE**.

### **Performance**

Finally, the size of a cookie can have an impact on the speed of your site.

## **Using sessions in views-**

When **SessionMiddleware** is activated, each **HttpRequest** object – the first argument to any Django view function – will have a **session** attribute, which is a dictionary-like object.

You can read it and write to **request.session** at any point in your view. You can edit it multiple times.

### **class backends.base.SessionBase**

This is the base class for all session objects. It has the following standard dictionary methods:

**\_\_getitem\_\_(key)**

Example: **fav\_color = request.session['fav\_color']**

**\_\_setitem\_\_(key, value)**

Example: **request.session['fav\_color'] = 'blue'**

**\_\_delitem\_\_(key)**

Example: **del request.session['fav\_color']**. This raises **KeyError** if the given **key** isn't already in the session.

**\_\_contains\_\_(key)**

Example: **'fav\_color' in request.session**

**get(key, default=None)**

Example: **fav\_color = request.session.get('fav\_color', 'red')**

**pop(key, default=\_\_not\_given)**

Example: **fav\_color = request.session.pop('fav\_color', 'blue')**

**keys()**

---

**items()**  
**setdefault()**  
**clear()**

It also has these methods:

**flush()**

Deletes the current session data from the session and deletes the session cookie. This is used if you want to ensure that the previous session data can't be accessed again from the user's browser (for example, the **django.contrib.auth.logout()** function calls it).

**set\_test\_cookie()**

Sets a test cookie to determine whether the user's browser supports cookies. Due to the way cookies work, you won't be able to test this until the user's next page request. See [Setting test cookies](#) below for more information.

**test\_cookie\_worked()**

Returns either **True** or **False**, depending on whether the user's browser accepted the test cookie. Due to the way cookies work, you'll have to call **set\_test\_cookie()** on a previous, separate page request. See [Setting test cookies](#) below for more information.

**delete\_test\_cookie()**

Deletes the test cookie. Use this to clean up after yourself.

**get\_session\_cookie\_age()**

Returns the value of the setting **SESSION\_COOKIE\_AGE**. This can be overridden in a custom session backend.

**set\_expiry(*value*)**



---

Sets the expiration time for the session. You can pass a number of different values:

- If **value** is an integer, the session will expire after that many seconds of inactivity. For example, calling **request.session.set\_expiry(300)** would make the session expire in 5 minutes.
- If **value** is a **datetime** or **timedelta** object, the session will expire at that specific date/time. Note that **datetime** and **timedelta** values are only serializable if you are using the **PickleSerializer**.
- If **value** is **0**, the user's session cookie will expire when the user's web browser is closed.
- If **value** is **None**, the session reverts to using the global session expiry policy.

Reading a session is not considered activity for expiration purposes. Session expiration is computed from the last time the session was *modified*.

### **get\_expiry\_age()**

Returns the number of seconds until this session expires. For sessions with no custom expiration (or those set to expire at browser close), this will equal **SESSION\_COOKIE\_AGE**.

This function accepts two optional keyword arguments:

- **modification**: last modification of the session, as a **datetime** object. Defaults to the current time.
- **expiry**: expiry information for the session, as a **datetime** object, an **int** (in seconds), or **None**. Defaults to the value stored in the session by **set\_expiry()**, if there is one, or **None**.

### **Note**

This method is used by session backends to determine the session expiry age in seconds when saving the session. It is not really intended for usage outside of that context.

In particular, while it is **possible** to determine the remaining lifetime of a session **just when** you have the correct **modification** value **and** the **expiry** is set as a **datetime** object, where you do have the **modification** value, it is more straight-forward to calculate the expiry by-hand:

```
expires_at = modification + timedelta(seconds=settings.SESSION_COOKIE_AGE)
```

### **get\_expiry\_date()**

Returns the date this session will expire. For sessions with no custom expiration (or those set to expire at browser close), this will equal the date **SESSION\_COOKIE\_AGE** seconds from now.

This function accepts the same keyword arguments as **get\_expiry\_age()**, and similar notes on usage apply.

### **get\_expire\_at\_browser\_close()**

Returns either **True** or **False**, depending on whether the user's session cookie will expire when the user's web browser is closed.

### **clear\_expired()**

Removes expired sessions from the session store. This class method is called by **clearsessions**.

### **cycle\_key()**

Creates a new session key while retaining the current session data. **django.contrib.auth.login()** calls this method to mitigate against session fixation.

---

## Working with JSON and AJAX

### Django JsonResponse example :

This example demonstrates how to send JSON data in Django

#### Step 1) open command prompt and execute the following commands

```
> mkdir jsonresponse  
> cd jsonresponse  
> mkdir src  
> cd src
```

We create the project and the and src directories. Then we locate to the src directory.

#### Step 2) run the command

```
> django-admin startproject jsonresponse .
```

We create a new Django project in the src directory.

Note: If the optional destination is provided, Django will use that existing directory as the project directory. If it is omitted, Django creates a new directory based on the project name. We use the dot (.) to create a project inside the current working directory.

We locate to the project directory.

**Step 3) open command prompt and execute the following command to show the tree structure**

```
> tree /f
src
|  manage.py
|
└── jsonresponse
    settings.py
    urls.py
    views.py
    wsgi.py
    __init__.py
```

- **Note:** The Django way is to put functionality into apps, which are created with `django-admin startapp`. In this tutorial, we do not use an app to make the example simpler. We focus on demonstrating how to send JSON response

**Step 4 ) open file `src/jsonresponse/urls.py` and add following code**

```
from django.contrib import admin
from django.urls import path
from .views import send_json
urlpatterns = [
    path('admin/', admin.site.urls),
    path('sendjson/', send_json, name='send_json'),
]
```

We will add a new route page; it calls the `send_json()` function from the `views.py` module.

**Step 5 ) create new `views.py` under `src/jsonresponse/` and add following code**

```

from django.http import JsonResponse

def send_json(request):

    data = [{'name': 'Peter', 'email': 'peter@example.org'},

            {'name': 'Julia', 'email': 'julia@example.org'}]

    return JsonResponse(data, safe=False)

```

Inside `send_json()`, we define a list of dictionaries. Since it is a list, we set `safe` to `False`. If we did not set this parameter, we would get a `TypeError` with the following message:

In order to allow non-dict objects to be serialized set the `safe` parameter to `False`.

### Step 6 ) open command prompt and run the command

```
> python manage.py runserver
```

- Step 7) We run the server and navigate to <http://127.0.0.1:8000/sendjson/>
- Step 8) We use the curl tool to make the GET request .open command prompt and run command  

```
> curl localhost:8000/sendjson/
```

**It shows output as follows:**

```
[{"name": "Peter", "email": "peter@example.org"},
{"name": "Julia", "email": "julia@example.org"}]
```

## Working with AJAX

---

## What is AJAX

- AJAX stands for Asynchronous JavaScript And XML, which allows web pages to update asynchronously by exchanging data to and from the server.
- This means you can update parts of a web page without reloading the complete web page.
- It involves a combination of a browser built-in XMLHttpRequest object, JavaScript, and HTML DOM.

## How AJAX Works –

1. An event occurs on a web page, such as an initial page load, form submission, link or button click, etc.
  2. An XMLHttpRequest object is created and sends the request to the server .
  3. The server responds to the request.
  4. The response is captured and then server respond back with response data.
- There are many scenarios where you may want to make GET and POST requests to load and post data from the server asynchronously, back and forth. Additionally, this enables web applications to be more dynamic and reduces page load time.

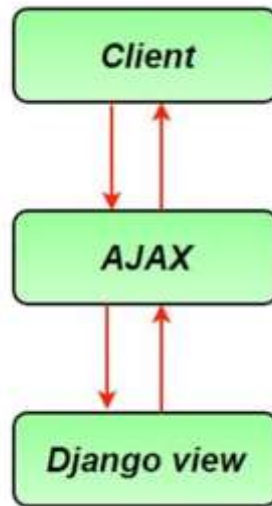


Image 48: Python- Working of AJAX

Source: <https://media.geeksforgeeks.org/wp-content/uploads/ajax.jpg>

## References

1. Introduction to the Internet by Scott D.James
2. <https://www.w3schools.com/html/>
3. <https://www.w3docs.com/learn-html.html>
4. <https://www.tutorialspoint.com/html/index.htm>
5. <https://www.educba.com/what-is-css/>
6. [https://www.tutorialspoint.com/css/css\\_tutorial.pdf](https://www.tutorialspoint.com/css/css_tutorial.pdf)
7. <https://code.tutsplus.com/tutorials/10-css3-properties-you-need-to-be-familiar-with-net-16417>
8. <https://www.hostinger.in/tutorials/difference-between-inline-external-and-internal-css>
9. <https://www.javatpoint.com/inline-css>
10. <https://www.w3schools.in/bootstrap4/intro/>
11. <https://www.uplers.com/blog/what-are-the-pros-cons-of-foundation-and-bootstrap/>
12. <https://azure.microsoft.com/en-in/overview/what-is-cloud-computing/#uses>
13. <https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html>
14. <https://www.ibm.com/in-en/cloud/learn/cloud-computing>
15. <https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html>
16. <https://aws.amazon.com/about-aws/global-infrastructure/>
17. <https://www.eztalks.com/webinars/what-is-a-live-webinar-and-how-does-it-work.html>
18. <https://www.ventureharbour.com/webinar-software-10-best-webinar-platforms-compared/>
19. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4432776/>
20. <https://www.gatevidyalay.com/types-of-attributes/>
21. <https://codeandwork.github.io/courses/cs/sqlErdSimpleQueries.html>
22. <https://www.csetutor.com/er-diagram-in-dbms-concept-importance-example/>



- 
23. <https://www.javatpoint.com/dbms-keys>
  24. <https://www.guru99.com/dbms-keys.html#7>
  25. <https://www.javatpoint.com/unique-key-in-sql>
  26. <https://www.javatpoint.com/dbms-mapping-constraints>
  27. <https://www.studytonight.com/dbms/er-diagram.php>
  28. <https://tutorialwing.com/mapping-constraints-in-dbms-for-relationship-types/>
  29. <https://prepinsta.com/dbms/mapping-constraints/>
  30. [https://docs.oracle.com/cd/B28359\\_01/server.111/b28318/part\\_3.htm](https://docs.oracle.com/cd/B28359_01/server.111/b28318/part_3.htm)
  31. <https://dev.mysql.com/doc/refman/8.0/en/features.html>
  32. <https://docs.mongodb.com/manual/reference/command/features/>
  33. <https://www.microsoft.com/en-us/sql-server/sql-server-2017-features>
  34. <https://www.studytonight.com/>
  35. <https://www.tutorialspoint.com/>
  36. <https://www.guru99.com/>
  37. <https://www.javatpoint.com/>
  38. <https://www.geeksforgeeks.org/>
  39. [https://www.w3schools.com/php/php\\_if\\_else.asp](https://www.w3schools.com/php/php_if_else.asp)
  40. <https://www.geeksforgeeks.org/php-decision-making/>
  41. <https://www.geeksforgeeks.org/php-loops/?ref=lbp>
  42. <http://shubhneet.com/mixing-decisions-and-looping-with-html/>
  43. [https://www.tutorialspoint.com/php/php\\_functions.htm](https://www.tutorialspoint.com/php/php_functions.htm)
  44. <https://www.c-sharpcorner.com/UploadFile/d9da8a/call-by-value-and-call-by-reference-in-php/>
  45. <https://www.splessons.com/lesson/php-functions/>
  46. [https://www.w3schools.in/php/functions/#PHP\\_Default\\_Argument\\_Value](https://www.w3schools.in/php/functions/#PHP_Default_Argument_Value)
  47. <https://www.javatpoint.com/php-recursive-function>
  48. <https://www.javatpoint.com/php-string-functions>
  49. <http://shubhneet.com/creating-and-accessing-string/>

- 
50. [https://www.javatpoint.com/php-string-str\\_replace-function](https://www.javatpoint.com/php-string-str_replace-function)
  51. <https://www.elated.com/formatting-php-strings-printf-sprintf/>
  52. [https://www.w3schools.com/php/php\\_string.asp](https://www.w3schools.com/php/php_string.asp)
  53. [https://www.tutorialspoint.com/php/php\\_arrays.htm](https://www.tutorialspoint.com/php/php_arrays.htm)
  54. <https://www.geeksforgeeks.org/php-arrays/>
  55. <https://www.studytonight.com/php/php-array-functions>
  56. [http://dsl.org/cookbook/cookbook\\_8.html](http://dsl.org/cookbook/cookbook_8.html)
  57. <https://www.geeksforgeeks.org/>
  58. [https://www.tutorialspoint.com/php/php\\_file\\_uploading.htm](https://www.tutorialspoint.com/php/php_file_uploading.htm)
  59. <https://www.tutorialrepublic.com/php-tutorial/php-file-download.php>
  60. [https://www.w3schools.com/php/php\\_mysql\\_intro.asp](https://www.w3schools.com/php/php_mysql_intro.asp)
  61. <https://www.tutorialspoint.com/mysql/mysql-connection.htm>
  62. <https://beginnersbook.com/2018/01/introduction-to-python-programming/>
  63. [https://www.tutorialspoint.com/python/python\\_overview.htm](https://www.tutorialspoint.com/python/python_overview.htm)
  64. <http://net-informations.com/python/iq/objects.htm>
  65. <https://elearningindustry.com/advantages-of-python-programming-languages>
  66. <https://www.javatpoint.com/python-history>
  67. <https://www.faceprep.in/python/python-installation/>
  68. <https://www.python.org/downloads/>
  69. <https://docs.python-guide.org/starting/install3/linux/>
  70. <https://www.edureka.co/blog/add-python-to-path/>
  71. [https://www.tutorialspoint.com/python/python\\_basic\\_syntax.htm](https://www.tutorialspoint.com/python/python_basic_syntax.htm)
  72. <https://www.pythonforbeginners.com/basics/python-syntax-basics>
  73. <https://www.programiz.com/python-programming/operators>
  74. <https://www.geeksforgeeks.org/decision-making-python-else-nested-elif/>
  75. <https://www.faceprep.in/python/loops-in-python/>
  76. <https://www.geeksforgeeks.org/loops-and-loop-control-statements-continue-break-and-pass-in-python/>
  77. Reference1 : <https://docs.python.org/3/tutorial/>

78. Reference 2 : <https://intellipaat.com/blog/tutorial/python-tutorial/>
79. Reference 3 : <https://www.techbeamers.com/python-programming-tutorials/>
80. Reference 4 : <https://www.programiz.com/python-programming>
81. Reference 5: <https://realpython.com/>
82. Reference 6: <https://www.phptpoint.com/python-tutorial/>
83. Reference 7: <https://www.guru99.com/python-tutorials.html>
84. Reference 8 : <https://www.datacamp.com/>
85. Reference 9 : <https://www.learnpython.org>
86. Web framework [https://en.wikipedia.org/wiki/Web\\_framework](https://en.wikipedia.org/wiki/Web_framework)
87. What is Django [https://en.wikipedia.org/wiki/Django\\_\(web\\_framework\)](https://en.wikipedia.org/wiki/Django_(web_framework))
88. MVC [https://www.tutorialspoint.com/struts\\_2/basic\\_mvc\\_architecture.htm](https://www.tutorialspoint.com/struts_2/basic_mvc_architecture.htm)
89. MVT <https://www.geeksforgeeks.org/django-project-mvt-structure/#:~:text=Django%20is%20based%20on%20MVT,for%20developing%20a%20web%20application.&text=View%3A%20The%20View%20is%20the,CSS%2FJavascript%20and%20Jinja%20files.>
90. Views <https://docs.djangoproject.com/en/4.0/topics/http/views/>
91. <https://www.geeksforgeeks.org/views-in-django-python/>
92. SQL <https://docs.djangoproject.com/en/4.0/topics/db/sql/>
93. Sessions : <https://docs.djangoproject.com/en/4.0/topics/http/sessions/AJAX:https://www.pluralsight.com/guides/work-with-ajax-Django>
94. <https://www.guru99.com/django-tutorial.html#5>
95. <https://www.javatpoint.com/django-tutorial>
96. Request and Response objects <https://docs.djangoproject.com/en/4.0/ref/request-response/>
97. Get and post method <https://docs.djangoproject.com/en/4.0/topics/forms/#:~:text=GET%20and%20POST&text=Django's%20login%20form%20is%20returned,this%20to%20compose%20a%20URL>
98. Django Tutorial1 <https://docs.djangoproject.com/en/4.0/intro/tutorial01/>
99. Django Tutorial2 <https://docs.djangoproject.com/en/4.0/intro/tutorial02/>