BACKEND WITH PYTHON (DJANGO)

Univelcity

Introduction

Programming is the art of solving computable problems.

Algorithms are processes or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Web development is mainly divided into two parts i.e frontend and backend. The frontend is all that the user sees i.e. the visuals and the designs. And the backend is all logic that makes things happen. In web development, all the functions you do like adding data and pressing the button are done on the frontend side, while whatever happened behind the scenes like collecting the data, processing it, encrypting it, adding that form data to the database would be done on the backend side.

Introduction(Cont’d)

Backend Development is also known as **server-side development**. It is everything that the users don’t see and contains behind-the-scenes activities that occur when performing any action on a website. It focuses primarily on databases, backend logic, APIs, and Servers.

The backend of a website is a combination of servers, applications, and databases. Code written by backend developers helps browsers in communicating with the databases and store data into the database, read data from the database, update the data and delete the data or information from the database.

Outline

- Variables
- Data Types
- Data Structures
- Data Conversion
- Built-in functions (len, map, input, random etc)
- Built-in modules (time, datetime and random)
- If/elif/else Statement
- Loops
- File I/0
- Functions
- OOP

Outline (contd.)

- Django and it's architecture
- Building applications with Django
- Understanding Http Methods and status/response codes
- Building RESTful APIs with Django

Python Variables

Variables are containers for storing data values.

Guidelines for naming variables

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (gender, Gender and GENDER are three different variables)
- A variable name cannot be an in-built python function. E.g print, type, list, bool, etc

Data Types

- Integers (int) - any whole number between -infinity to +infinity e.g 99
- Strings (str) - any character enclosed in single or double quotation e.g ‘good’, “blast” etc
- Floats (float) - decimal numbers e.g 9.4
- Boolean (bool) - True or False.

Data Structures

- Lists (list) - A collection of Data enclosed in square brackets [ ], having each entry comma separated.
  E.g [‘music’ , “sports”, True, 99.9]
  They are ordered and changeable. They allow duplicate values.

- Tuples (tuple) - A collection of Data enclosed in parenthesis ( ), having each entry comma separated.
  E.g (‘music’ , “sports”, True, 99.9)
  They are ordered and unchangeable. They allow duplicate values.

Data Structures (contd.)

- Sets (set) - A collection of Data enclosed in curly braces { }, having each entry comma separated.
  E.g {‘music’ , “sports”, True, 99.9}
  They are unordered and changeable. They do not allow duplicates.

- Dictionaries (dict) - A collection of Data enclosed in curly braces { }, having each entry comma separated. Each data entry is a key-value pair having each pair colon separated.
  E.g {“Name”: “Adamu”, “Gender”: “Male”, “Age”: 12}
  Note: All keys can be strings, boolean, integer, float or tuple while the values can be any data type or data structure.

Operators

Arithmetic Operators (on int and float)

Operator	**Name**	**Example**
+	Addition	5 + 5 = 10
-	Subtraction	10 - 40 = -30
*	Multiplication	2 * 9 = 18
/	Division	74 / 2 = 37
%	Modulus	34 % 5 = 4
//	Floor Division	23 // 4 = 5
**	Exponential	6 ** 3 = 216

Operators (contd.)

Assignment Operators

Operator	**Example**	**Same As**
=	a = 20	a = 20
+=	a += 10	a = a + 10
-=	a -= 10	a = a - 10
*=	a *= 2	a = a * 2
/=	a /= 2.5	a = a / 2.5
%=	a %= 3	a = a % 3
//=	a //= 11	a = a // 11
**=	a **= 4	a = a ** 4

Operators (contd.)

Comparison Operators

Operator	**Name**	**Example**
==	Equal (the same as)	a == b
!=	Not equal (not the same as)	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

Operators (contd.)

Logical operators

Operator	Description	Example
or	Returns True if one of the statements is true	a < 29 or a > 54
and	Returns True if both statements are true	a > 5 and a > 15
not	Reverse the result, returns False if the result is true	not(a >= 19 and a < 38)

Operators (contd.)

Identity operators

Operator	Description	Example
is	Returns True if both variables are the same object	p is q
is not	Returns False if both variables are not the same object	p is not q

Operators (contd.)

Membership operators

Operator	Description	Example
in	Returns True if the specified value is present in the object	a in b
not in	Returns False if the specified value is present in the object	a not in b

Strings

Concatenation

Strings can be concatenated using the + operator. E.g “beautiful” + “city”

Indexing

String characters can be indexed (specifying the position). The position is enclosed in square brackets [ ].

E.g new_word = “Strength”

    print(new_word[3])

Negative Indexing

A string can be indexed using a negative number. The last character in the string is indexed -1, second to the last is indexed -2 and so on.

Strings (contd.)

Slicing

String characters can be sliced (cutting out a portion). The general syntax for slicing is [start : end : step].

E.g new_string = “blue sky”

```
print(new_string[2: 5])
print(new_string[: 5])
print(new_string[2: ])
print(new_string[2: 8: 2])
```

Strings (contd.)

Formatting

This is used to include;

- A string in another string sequence
- An integer in a string sequence
- A float in a string sequence
- A boolean in a string sequence

Strings (contd.)

Formatting contd.

There are two ways of formatting strings

- mini_string = “blacklist”
  max_string = “The {} is getting tiresome”.format(mini_string)
  print(max_string)

- mini_string = “blacklist”
  max_string = f“The {mini_string} is getting tiresome”
  print(max_string)

Strings (contd.)

Escape characters

An escape character is used to treat the succeeding character as normal.

E.g mod_string = ‘The Nation\’s crude oil’

```
print(mod_string)
```

Strings (contd.)

String methods

Method	Description
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case
startswith()	Returns true if the string starts with the specified value
endswith()	Returns true if the string ends with the specified value
index()	Searches the string for a specified value and returns the position of where it was found
find()	Searches the string for a specified value and returns the position of where it was found
isspace()	Returns True if all characters in the string are whitespaces

Strings (contd.)

String methods contd.

Method	**Description**
islower()	Returns True if all characters in the string are lower case
isupper()	Returns True if all characters in the string are upper case
split()	Splits the string at the specified separator, and returns a list
join()	Joins the items of an iterable to the end of the string
count()	Returns the number of times a specified value occurs in a string
replace()	Returns a string where a specified value is replaced with a specified value
strip()	Returns a trimmed version of the string

List

Indexing/Slicing

This is used to access items in a list. It has the same syntax as discussed in the strings.

Changing data

Any desired entry in a list can be changed.

E.g my_list = [“football”, 150, True, “false”]

```
my_list[2] = “True”
print(my_list)
```

Join Lists

Two or more lists can be joined using the + operator.

E.g first_list = [3, 6, 7] and second_list = [8, 9, 10]

```
full_list = first_list + second_list
print(full_list)
```

List (contd.)

List methods

Method	Description
clear()	Removes all the items from the list
count()	Returns the number of items with the specified value
extend()	Add the items of a list (or any iterable), to the end of the current list
index()	Returns the index of the first item with the specified value
remove()	Removes the item with the specified value
append()	Adds an item at the end of the list
copy()	Returns a copy of the list
insert()	Adds an item at the specified position
sort()	Sorts the list

List (contd.)

List methods

Method	**Description**
pop()	Removes the item at the specified position
reverse()	Reverses the order of the list

Tuple (contd.)

The items in a tuple can be accessed by either indexing or slicing. Recall, the items in tuple are not changeable.

Two tuples can be combined using the + operator.

```
E.g first_tuple = (False, “True”, 100)
        second_tuple = (1, 4, 9)
        full_tuple = first_tuple + second_tuple
       print(full_tuple)
```

Tuple (contd.)

Tuple methods

Method	**Description**
index()	Searches the tuple for a specified value and returns the position of where it was found
count()	Returns the number of times a specified value occurs in a tuple

Set

Set methods

Method	**Description**
discard()	Removes the specified item. Does not raise an error if element doesn’t exist in the given set.
add()	Adds an item to the set
clear()	Removes all the items from the set
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others
copy()	Returns a copy of the set
remove()	Removes the specified item but raises an error when the specified element doesn't exist in the given set.
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set

Set (contd.)

Set methods

Method	Description
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	Inserts the symmetric differences from this set and another
pop()	Removes an item from the set

Dictionary

Changing values of a specified key.

```
E.g my_diction = {“Pet”: “Dog”, “Colour”: “Brown”}
    my_diction[“Colour”] = “Black”
    print(my_diction)
```

Dictionary (contd.)

Dictionary methods

Method	Description
copy()	Returns a copy of the dictionary
get()	Returns the value of the specified key
keys()	Returns a list containing the dictionary's keys
values()	Returns a list of all the values in the dictionary
items()	Returns a list containing a tuple for each key value pair
update()	Updates the dictionary with the specified key-value pairs
pop()	Removes the element with the specified key
clear()	Removes all the elements from the dictionary

Data Conversion

- string to integer
  E.g int(“100”)
- integer to string
  E.g str(900)
- string to bool
  E.g bool(“True”)
- bool to string
  E.g str(“False”)
- int to float
  E.g float(45)
- float to int
  E.g int(99.9)

Data Conversion (contd.)

- list to string
  E.g str([“high”, “low”])
- string to list
  E.g list(“yeah”)
- list to tuple
  E.g tuple([1, 2, 3, 4])
- tuple to list
  E.g list((“Cool”, 9.9, True))
- list to set
  E.g set([30, 50, 30, “justice”])
- set to list
  E.g list({“low”, “medium”, 101.3})

Data Conversion (contd.)

- set to tuple
  E.g tuple({12, 90, 12})
- tuple to set
  E.g set((True, “True”, 0.9))
- Only a paired tuple in a list can be converted to a dictionary
  E.g dict([(“Name”, “John”), (“Age”, 10)])
- A dictionary method .items() as discussed earlier, returns dict_items. This output can then be converted to a list.

Built-in functions

- input()

  This function allows interaction between the user and the python code.
  It creates room in the VS Code terminal to receive inputs from the user.

  E.g get_data = input(“Enter data here: ”)

  ```
  print(get_data)
  ```

Built-in functions (contd.)

- len()
  This returns the length of any iterable i.e string, list, tuple, set etc
  E.g sports = {“swimming”, “boxing”, “cricket”}
  ```
  sports_count = len(sports)
  print(sports_count)
  ```

- sum()
  This adds up all the items in an iterable(usually numbers) and returns the total.
  E.g scores = [50, 60, 70, 80]
  ```
  print(sum(scores))
  ```

Built-in functions (contd.)

- min()
  This returns the minimum number in an iterable.
  E.g
  ```
  scores = [45, 78, 66, 98]
  print(min(scores))
  ```

- max()
  This returns the maximum number in an iterable.
  E.g
  ```
  scores = (65, 76, 35, 89)
  print(max(scores))
  ```

Built-in functions (contd.)

- zip()
  This takes in two iterables and pairs the items in the iterable together.
  E.g first_iterble = [2, 4, 6, 8]
  ```
  second_iterable = [“a”, “b”, “c”, “d”]
  pre_output = zip(first_iterable, second_iterable)
  output = list(pre_output)
  print(output)
  ```

  If both iterables are of different length, it will return as many zipped items as there are items in the smaller iterable.

  To see the zipped object, convert the variable to a list.

Built-in functions (contd.)

- enumerate()

  This functions zips a counter with the items in an iterable.

  E.g

  ```
  artists = [“Ed Sheeran”, “Usher”, “Oxlade”]
  output = enumerate(artists)
  print(list(output))
  ```

  Also, to see the enumerate object, convert the variable to a list.

Built-in functions (contd.)

- lambda

  A lambda function is an anonymous function which can have as many arguments as desired but only one expression.

  Its syntax is;   lambda *argument*: *expression*

  E.g

  ```
  my_adder = lambda x : x + 100
  prnt(my_adder(1))
  ```

  E.g

  ```
  my_math = lambda x, y, z : x **2 + y + z
  print(my_math(5, 6, 7))
  ```

Built-in functions (contd.)

- map()

  This executes a specified function for each item in an iterable.

  Its syntax is; map(*function*, *iterable*)

  E.g
  ```
  mapped_obj = map(lambda x : x + 70, [20, 40, 60])
  print(list(mapped_obj))
  ```

  Similarly, to view the mapped object, convert it to a list.

Built-in functions (contd.)

- filter()
  This filters an iterable in accordance with a specified function. It returns a list of the items that returned True in that function.

  E.g
  ```
  num = [20, 31, 45, 60 10, 77]
  my_filter = filter(lambda x : x % 2, num)
  print(list(my_filter))
  ```

  Similarly, it returns a filter object which can be viewed by converting to a list.

Built-in functions (contd.)

- range()
  This function returns a sequence of numbers.
  Its general sequence is range(*start*, *stop*, *step*).

  E.g

  ```
  my_num = range(1, 8)
  print(list(my_num))
  ```

  Start is optional. It starts at 0 by default

  Stop is required.

  Step is optional. It is 1 by default.

Built-in modules

time

time.sleep() - This halts the code for the specified number of seconds.

E.g import time

```
    print(“Hello there!”)
    time.sleep(5)
    print(“Done!”)
```

Built-in modules(contd.)

random
To use the random module, it needs to be imported.
E.g import random

- random.shuffle($iterable$): This shuffles the items in an iterable.
- random.choice($iterable$): This randomly selects an item in an iterable.
- random.sample($iterable$, k): This returns a list of k randomly selected items from an iterable. k, which is an integer, specifies the number of items that will be randomly selected.
- random.randrange($start$, $stop$, $step$): This returns a list of randomly selected integers with the start and stop range.
- random.seed($integer$): When defined, this seeds the random selection process to the particular integer passed as the argument. This means that if the program runs again, the random selection will be the same as the previous run.

Built-in modules(contd.)

datetime

Similarly, to use the module, it needs to be imported.

E.g from datetime import datetime

- datetime.now(): This returns the current date and time.
- datetime.now().hour: This returns the current hour. Similarly, minutes, month, year and seconds can be accessed.

Built-in modules(contd.)

datetime.strftime(): This can take a number of arguments as summarized below.

Argument	Description	Example
%A	Weekday full version	Thursday
%a	Weejday short version	Thur
%d	Day of the month	15
%B	Month name full version	February
%b	Month name short version	Feb
%Y	Year full version	2021
%y	Year short version	21
%x	Local version of date	15/02/21

If/elif/else statement

- This statement allows for a block of code to run if and only if certain conditions are met.
- The conditions can be a based on comparison operators, identity operators, membership operators or a boolean.

```
E.g if 77 > 77:
            print(“Yes”)
      else:
            print(“No”)
```

If/elif/else statement (contd.)

- When having to check for multiple conditions for the same block of code, the words **and/or** are used to merge the conditions.

  E.g
  ```
  if “city” == “city” and 45 > 9:
          print(“conditions met!”)
      else:
          print(“Oops!”)
  ```
- When using **and**, both/all conditions have to be met while using **or**, only one of the conditions has to be met.

If/elif/else statement (contd.)

- To check for two or more conditions with each condition/set of conditions having unique block of codes, use **elif** to specify those conditions.

  E.g if 50 != 50:

  ```
          print(“first condition met”)
      elif 51 // 5 = 10:
          print(“second condition met”)
      else:
          print(“none was met”)
  ```

Loops

Loops are a way of iterating a block of code.
There are two types of loops; **while** loops and **for** loops.

- **while** loop repeats a block of code as long as a condition (the result being a boolean) is being met.
- **for** loop repeats a block of code a specified number of times and the actions can also be on each element of an iterable.

while loop

- The variable on which the condition will be based should be defined prior to the while statement
- A while loop can be immediately terminated using the **break** statement

E.g
```
cracker = 0
    while (cracker < 5):
        print(“Cracker is less than 5”)
        cracker += 1
```

while loop (contd.)

E.g
```
count = 0
while  True:
    if count == 3:
        break
    else:
        print(“Music’s great!”)
        count += 1
```

for loop

- It repeats a block of code a specified number of times (as many items in the iterable).

```
E.g entry_data = “Universe”
      for a in entry_data:
            print(“Home for all!”)
```

```
E.g  new_list = [“pop”, “rock”, “country”]
       for item in new_list:
             print(item)
```

for loop with enumerate()

This serves as a means of keeping counts of each iteration.

E.g

```
names = [“Sheldon”, “Penny”, “Howard”, “Leonard”, “Rajesh”]
    for index, item in enumerate(names):
        print(index, item)
```

List comprehension

This is a means of writing a for loop in a single line of code.

Its general syntax is;

[*expression* for *member* in *iterable*]

With an if statement, its general syntax becomes;

[*expression* for *member* in *iterable* if condition]

E.g [print(a) for a in range(100, 161)]

File I/O

This entails interaction between our python codes and files (text, csv or excel).

This is based on a built-in function **open()**.

E.g new_file = open(“$Filepath + filename$”, mode = “r”)

    print(new_file.read())

    new_file.close()

File I/0 (contd.)

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist. This is the default mode.
- “r+” - Opens a file for both reading and writing.
- "a" - Append - Opens a file for appending, creates the file if it does not exist
- “a+” - Opens a file for both appending and reading.
- "w" - Write - Opens a file for writing, creates the file if it does not exist
- “w+” - Opens a file for both writing and reading.
- "x" - Create - Creates the specified file, returns an error if the file exists

File methods : write(), writelines(), read(), readline() and close()

Functions

Functions are segmentation of blocks of codes which will run only when called.

- Functions are defined using;

```
def functionname():
    Block of code
```

- Functions are called using;

```
functionname()
```

Functions (contd.)

Parameters are defined in the parentheses when defining a function.

Arguments are passed in the parentheses when calling a function.

A **return** statement is used to return the output of a function. This differs from print(). **print()** only just prints out the output while **return** gets to save the output in a variable which can be used later in the program.

E.g

```
def my_adder(x):
        return 50 + x

  my_adder(10)
```

Functions (contd.)

The number of the arguments passed must equal the number of parameters defined.

Arbitrary arguments: This is used when the number of arguments to be passed in a function is unknown.

E.g
```
def func_one(*names):
        print(“Hello” + names[1])

func_one(“James”, “Harley”, “Stones”)
```

Functions (contd.)

Keyword argument: This is used to assign the arguments passed to variables.

E.g def func_two(fruit2, fruit1):

```
        print("My favourite is" + " " + fruit2)

    func_two(fruit1 = "cherry", fruit2 = "apple")
```

Arbitrary keyword argument: This is used when the number of keyword arguments to be passed is unknown.

E.g def func_three(**country):

```
        print(country["east"] + " "  + "is a country in Africa")

    func_three(south = "South Africa", east = "kenya")
```

Functions (contd.)

Default parameter: This assigns a default value for a specified parameter. This comes in handy in scenarios where no argument is passed.

E.g
```
def func_four(car = “Tundra”):
        print(“I just got a” + “ ” + car)

func_four()
```

OOP(Object Oriented Programming)

This is a method of grouping related attributes and behaviours of objects together.

class: This is the blueprint of the object.

object: This is an instantiation of a class.

E.g
```
class door:
        def open():
            print(“This opens the door”)
        def close():
            print(“This closes the door”)
        def slide():
            print(“This slides the door”)

d1 = door()
print(d1.slide())
```

OOP(Object Oriented Programming contd.)

Attributes can be considered as data types stored in variables.
There are two types of attributes; class attribute and instance attribute.

- class attributes are attributes defined for the whole class. i.e, they are not associated with instances of the class
- instance attributes are attributes defined in the instance method of a class.

OOP(Object Oriented Programming contd.)

Methods are simply functions defined in a class.
There are 3 types of methods; instance methods, class methods and static methods.

- instance method is a method that is associated with instances of the class.
  self must be passed in as a parameter when defining the method but will not be passed as an argument when calling the method.
  E.g class fruit:

```
            def grape(self):
                *Block of code*

      f_one = fruit()
```

OOP(Object Oriented Programming contd.)

.__init__() method: This is an instance method, similarly, it requires *self* to be passed. It is used to define instance attributes in the class.
Note: This methods does not require being called. It runs automatically.
E.g class car:

```
        def __init__(self, brand,  price):
            self.brand = brand
            self.price = price

        def get_price(self):
            return self.price


    c1 = car("Tundra", 8358000)
    print(c1.get_price())
```

OOP(Object Oriented Programming contd.)

- class method: This is a method where class attributes are defined. A decorator is needed in order to initiate a class method. cls needs to be passed in as a parameter when defining the function but it will not not be passed as an argument when calling the function.
  E.g class door:

```
        width = 4

        @classmethod
        def get_width(cls):
            return cls.width

        @classmethod
        def set_width(cls, value):
            cls.width = value

    d2 = door()
    print(d2.get_width())
```

OOP(Object Oriented Programming contd.)

- static method: This is used when changes are not to be made on a method or an attribute. In other words, this method prohibits change.
  Like the class method, a static method also requires a decorator.
  E.g class cruise:

```
            @staticmethod
            def info():
                  print(“This is a class for cruise and vibes!”)

     cru_vib = cruise()
     print(cru_vib.info())
```

Virtual Environments

A virtual environment is an isolated Python environment where a project's dependencies are installed in a different directory from those installed in the system's default Python path and other virtual environments.

Why use virtual environments?

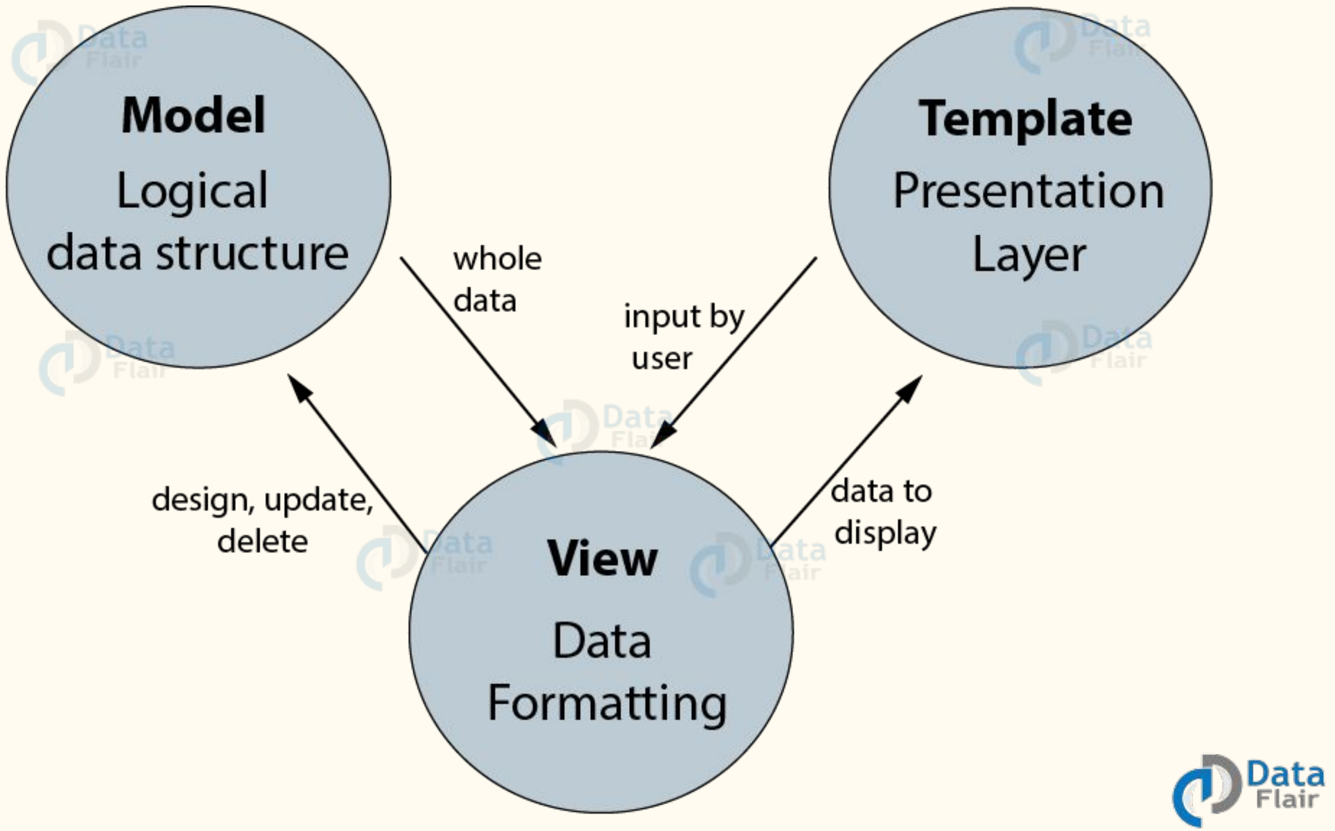
- Dependency managers are tools that enable easy management of a project's dependencies.
- The problems solved by virtual environments
- Managing projects with different dependencies. For example, when one project requires Django==2.6 but another project requires Django==3.0.6.
- Installation of Python packages and libraries.
- Dependency resolution. This is where you specify the requirements for a particular project's sub-dependency to avoid installation problems.
- Reproduction of environments. One machine can handling many projects in different environments.

Introduction to Django

Django is basically a high-level Python web application framework that enables the rapid development of web applications. It achieves so with pragmatic, much cleaner design and is also easy to use (in comparison of other frameworks) thus is very popular among web developers.

All the functionality comes in the Django framework in the form of web applications. You just have to import those applications according to your need and thus you can concentrate more on the unique application of your website rather than dealing with all these backend problems.

Django Architecture



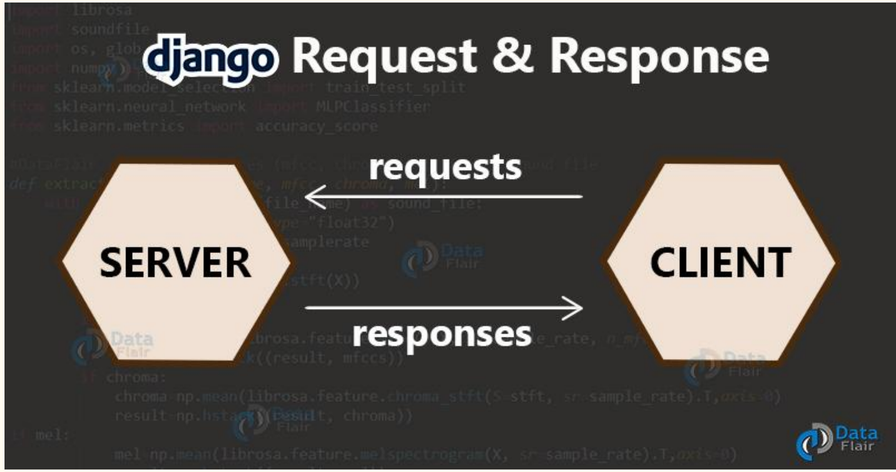
Django Architecture(Cont’d)

When we make a request for the website, the interface through which we use to make that request via our browser was the Template. Then that request transmits to the server for the management of view file.
Django is literally a play between the requests and responses. So whenever our Template is updating it’s the input (request) we sent from here which on the server was seen by the View. And, then it transports to the correct URL. It’s one of the important components of Django MTV architecture. There, the URL mapping in Django is actually done in regular expressions.
Now after the sending of request to the correct URL, the app logic applies and the model initiates to correct response to the given request. Then that particular response is sent back to the View where it again examines the response and transmits it as an HTTP response or desired user format. Then, it again renders by the browser via Templates.

HTTP Request and Response

These interchanges between various servers and client machines following a set of rules. That set of rules is called HTTP. HTTP stands for **Hyper Text Transfer Protocol**. It's simply a set of rules, a common standard followed by devices to connect to each other over the internet.

The information sent or received is given the name of request and response.
Request Objects contain the request made by the client. The server responds with the appropriate information according to Request Object.



HTTP Request and Response (Cont’d)

The requests are smaller in size than response. Basically, a Request is a “request made by the client” to the server. Any URL that gets searched is a request.
But, that’s not the only type of Request. Request objects also give information to the server. The forms we fill, the images we upload to the server are also Requests. When we upload this information, it is transferred as attributes of the Request object.
The server can then access that information and call its own functions.

HTTP Methods

- **GET:** The HTTP GET method is used to *read* (or retrieve) a representation of a resource. In case of success (or non-error), GET returns a representation in JSON and an HTTP response status code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

- **POST:** The POST method is most often utilized to *create* new resources. In particular, it is used to create subordinate resources. That is subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc. On successful creation, HTTP response code 201 is returned.

HTTP Methods(Cont'd)

- **PUT/PATCH :** PUT or PATCH is used to *modify* resources. The PATCH request only needs to contain the changes to the resource, not the complete resource. In other words, the body should contain a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. While the PUT request contains the changes to the complete resource.

- **DELETE :** DELETE is quite easy to understand. It is used to *delete* a resource identified by filters or ID. On successful deletion, the HTTP response status code 204 (No Content) returns with no response body.

HTTP Status Codes

The status codes are divided into the five categories.

- **1xx: Informational** – Communicates transfer protocol-level information.
- **2xx: Success** – Indicates that the client’s request was accepted successfully.
- **3xx: Redirection** – Indicates that the client must take some additional action in order to complete their request.
- **4xx: Client Error** – This category of error status codes points the finger at clients.
- **5xx: Server Error** – The server takes responsibility for these error status codes.