***Table of Content***

|  |  |  |
| --- | --- | --- |
| ***S.No*** | ***Topics*** | ***Page No.*** |
| 1 | Brief About Python. |  |
| 2 | PIP(Package Management Tool). |  |
| 3 | Virtualenv (Virtual Environment). |  |
| 4 | Fabric (Deployment Tool). |  |
| 5 | Django(Popular Python Framework). |  |
| 6 | Django templating . |  |
| 8 | Database Drivers Required. |  |
| 9 | South(Migration Tool). |  |
| 10 | Exceptions in Python |  |
|  |  |  |
|  |  |  |

***Python***

**Python History** -

* Python laid its foundation in the late 1980s.
* The implementation of Python was started in the December 1989 by **Guido Van Rossum** at CWI in Netherland.
* *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
* Python is influenced by programming languages like:
  + ABC language.
  + Modula-3.

**What is python ?**

* **Python** is an object-oriented, high level language, interpreted, dynamic and multipurpose programming language.
* Python is ***easy*** *to learn* yet powerful and versatile scripting language which makes it attractive for Application Development.
* Python's syntax and ***dynamic typing*** with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas.
* Python supports ***multiple programming pattern***, including object oriented programming, imperative and functional programming or procedural styles.
* Python is not intended to work on special area such as web programming. That is why it is known as ***multipurpose*** because it can be used with **web**, **enterprise**, **3D** **CAD** etc.
* We don't need to use data types to declare variable because it is ***dynamically******typed*** so we can write a=10 to declare an integer value in a variable.
* Python makes the development and debugging *fast* because there is no compilation step included in python development and edit-test-debug cycle is very fast.
* ***Python Features* *:-***

There are a lot of features provided by python programming language.

* Easy to Use - Python is easy to very easy to use and high level language. Thus it is programmer-friendly language.
* Expressive Language - Python language is more expressive. The sense of expressive is the code is easily understandable.
* Interpreted Language - Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.
* Cross platform Language - Python can run equally on different platforms such as Windows, Linux, Unix , Macintosh etc. Thus, Python is a portable language.
* Free and Open Source -Python language is freely available(www.python.org).The source-code is also available. Therefore it is open source.
* Object Oriented Language - Python supports object oriented language. Concept of classes and objects comes into existence.
* Extensible - It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in your python code.
* Large Standard Library - Python has a large and broad library.
* GUI Programming - Graphical user interfaces can be developed using Python.
* Integrated - It can be easily integrated with languages like C, C++, JAVA etc.

**How to Install :-**

If you’re on Linux or Mac OS X, you probably already have Python installed. Type python at a command prompt (or in Terminal, in OS X). If you see something like this, then Python is installed:

**Python 2.4.1 (#2, Mar 31 2005, 00:05:10)**

**[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin**

**Type "help", "copyright", "credits" or "license" for more information.**

**>>>**

Otherwise, if you see an error such as "command not found", you’ll have to download and install Python.

See <http://www.python.org/download/> to get started. The installation is fast and easy.

* **Mac OSX**

Complete below steps to get your computer setup to run Python. You should follow these instructions as exactly as possible. For example, the latest version of Mac OSX computers already have Python 2.7, so do not install.

1. Get the gedit text editor, and install it.

2. Put gedit (your editor) in your Dock so you can reach it easily.

(a) Run gedit so we can fix some stupid defaults it has.

(b) Open Preferences from the gedit menu and select the Editor tab.

(c) Change Tab width: to 4.

(d) Select (make sure a check mark is in) Insert spaces instead of tabs.

(e) Turn on “Automatic indentation” as well.

(f) Open the View tab and turn on “Display line numbers”.

3. Find your “Terminal” program. Search for it. You will find it.

4. Put your Terminal in your Dock as well.

5. Run your Terminal program. It won’t look like much.

6. In your Terminal program, run python. You run things in Terminal by just typing their name and hitting RETURN.

7. Hit CTRL-D (^D) and get out of python.

8. You should be back at a prompt similar to what you had before you typed python. If not find out why.

9. Learn how to make a directory in the Terminal. Search online for help.

10. Learn how to change into a directory in the Terminal. Again search online.

11. Use your editor to create a file in this directory. You will make the file, “Save” or “Save As...”, and pick this directory.

12. Go back to Terminal using just the keyboard to switch windows. Look it up if you can’t figure it out.

13. Back in Terminal, see if you can list the directory to see your newly created file. Search online for how to list a

directory.

* **Windows**

1. Get the gedit text editor, and install it. You do not need to be administrator to do this.

2. Make sure you can get to gedit easily by putting it on your desktop and/or in Quick Launch. Both options are available during setup.

(a) Run gedit so we can fix some stupid defaults it has.

(b) Open Edit->Preferences select the Editor tab.

(c) Change Tab width: to 4.

(d) Select (make sure a check mark is in) Insert spaces instead of tabs.

(e) Turn on “Automatic indentation” as well.

(f) Open the View tab turn on “Display line numbers”.

3. Find your “Terminal” program. It’s called Command Prompt. Alternatively just run cmd.

4. Make a shortcut to it on your desktop and/or Quick Launch for your convenience.

5. Run your Terminal program. It won’t look like much.

6. In your Terminal program, run python. You run things in Terminal by just typing their name and hitting

RETURN.

(a) If you run python and it’s not there (python is not recognized..).

http://python.org/download

Install it from

(b) Make sure you install Python 2.7.

(c) You may be better off with ActiveState Python especially when you miss Administrative rights

7. Hit CTRL-Z (^Z), Enter and get out of python.

8. You should be back at a prompt similar to what you had before you typed python. If not find out why.

9.**Warning**: Windows is a big problem for Python. Sometimes you install Python and one computer will have no problems, and another computer will be missing important features. If you have problems, please visit:

<http://docs.python.org/faq/windows.html>

10.**On Windows,** you’ll need to update your PATH environment variable.

Refer the below links for more details : -

*Downloading and installing Python and Django on Windows*

<http://www.youtube.com/watch?v=rIVwVOpwpsA>

*Configuring : -*

<http://www.youtube.com/watch?v=NqNNuXwi0eI>

* **Linux**

**If yoursudo apt-get install python2.7**

1. Get the gedit text editor, and install it.

2. Make sure you can get to gedit easily by putting it in your window manager’s menu.

(a) Run gedit so we can fix some stupid defaults it has.

(b) Open Preferences select the Editor tab.

(c) Change Tab width: to 4.

(d) Select (make sure a check mark is in) Insert spaces instead of tabs.

(e) Turn on “Automatic indentation” as well.

(f) Open the View tab turn on “Display line numbers”.

3. Find your “Terminal” program. It could be called GNOME Terminal, Konsole, or xterm.

4. Put your Terminal in your Dock as well.

5. Run your Terminal program. It won’t look like much.

6. In your Terminal program, run python. You run things in Terminal by just typing their name and hitting RETURN. a. If you run python and it’s not there, install it. Make sure you install Python 2.7.

7. Hit CTRL-D (^D) and get out of python.

8. You should be back at a prompt similar to what you had before you typed python. If not find out why.

9. Learn how to make a directory in the Terminal. Search online for help.

10. Learn how to change into a directory in the Terminal. Again search online.

11. Use your editor to create a file in this directory. Typically you will make the file, “Save” or “Save As..”, and pick this directory.

12. Go back to Terminal using just the keyboard to switch windows. Look it up if you can’t figure it out.

**How to Execute Python ?**

There are three different ways of working in Python:

* **Interactive Mode -** You can enter python in the terminal and start working with Python. open terminal type “python” and hit enter.

Python 2.7.3 (default, Feb 27 2014, 19:58:35)

[GCC 4.6.3] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>

Now you can execute your Python commands.

>>> a=10

>>> b=20

>>> c=a+b

>>> print c

30

>>>

* **Script Mode -** Using Script Mode , you can write your Python code in a separate file using any editor of your Operating System.

Create a file write your code Save it by .py extension.

Open the terminal execute it by “python filename.py” you will get the output of code

on terminal.

* **Using IDE** (Integrated Development Environment) - There are lot of IDE’s available for python the most popular is pycharm you can Download and use it from <https://www.jetbrains.com/pycharm/>

***PIP***

**What is PIP ?**

* A tool for installing and managing Python packages .pip is a package management system used to install and manage [software packages](http://en.wikipedia.org/wiki/Package_(package_management_system)) written in Python. Many packages can be found in the [Python Package Index](http://en.wikipedia.org/wiki/Python_Package_Index) .
* One major advantage of pip is the ease of its [command-line interface](http://en.wikipedia.org/wiki/Command-line_interface), which makes installing Python software packages as easy as issuing one command:

pip install some-package-name

* Users can also easily implement the package's subsequent removal:

pip uninstall some-package-name

* Most importantly pip has a feature to manage full lists of packages and corresponding version numbers.

**Why to use PIP ?**

**Python and Packages** :-

Although Python applications can be made of a single file, usually they consist of a series of functions, objects (classes), handy tools and of course, variables spread across multiple file(s), placed inside *modules*. These modules together make up what is referred as a *package*.

The traditional way of installing a package involves first spotting it and then downloading. It sounds soft and simple because it actually *is* like many things in Python - *but it is not perfect*.

When the files are ready and unpacked, using the *distutils* module, you can install it by calling setup.py

**cd a\_package  
python setup.py install**

In spite of the simplicity of the procedure explained above, it is no use if the challenge abstracted from installing exists elsewhere in the process: finding and managing them. This is where package management via tools comes in - bringing along several benefits such as:

* Uninstalling (e.g. pip uninstall package\_x),
* Versioning (e.g. pip install django==x),
* And automatic dependency management (as packages can depend on others).

### Package Management

Packages in Python can be tools, libraries, frameworks and applications. Given the popularity and the beauty of the language, there are tens of thousands of packages available which you can make use of for *your own projects*.

### Package Management Tools

The two most common Python package managers are *pip* and *easy*install\_. Both of them aim to help users with the tasks of:

* Downloading, Installing and Uninstalling
* Building
* Managing Python packages and much more

Both of them might appear to do the same thing from the outside and their joint dependence on the common library *setuptools* increases this notion.

However, in this case, it is what's hidden from the eye that makes the difference — and a lot of it as well.

How to install ?

We are going to securely download the setup files for setuptools using curl. cURL is a system library which allows data transfer over various protocols (i.e. a common language for data exchange between applications, such as HTTP). It will verify the *SSL* certificates from the source and pass the data to the *Python interpreter*.

These setup files, which Python interpreter is going to execute, automate the installation process as they set up the latest stable version on our system.

Execute the following command:

**curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez\_setup.py | python -**

This installation gives us the ability to use pip globally across the system. However, this is not the preferred way to install any other package. What is recommended is to always use self-contained Python environments, virtualenv. We will talk about it in the next section.

Note: You might need to explicitly gain *superuser* privileges in order to continue with the download. In that case, consider using:

**curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez\_setup.py | sudo python -**

***Virtualenv(Virtual Environment)***

**What is Virtualenv ?**

Let's begin with defining what exactly virtualenv is and the situation where it comes in handy.

In the world of Python, an *environment* is a folder (directory) which contains everything that a Python project (application) needs in order to run in an organised, isolated fashion. When it is initiated, it automatically comes with its own Python interpreter - a copy of the one used to create it - alongside its very own *pip*.

virtualenv allows you to work on a specific project without worry of affecting other projects It enables multiple side-by-side installations of Python, one for each project.

**Why to use ?**

There are a number of problems that virtualenv solves:

* Creating a fresh, isolated environment for a Python project
* Being able to download packages without requiring admin/sudo privileges
* Easily packaging an application
* Creating a list of dependencies which belongs to a single project created with *pip*
* Easily recovering the dependencies using a requirements file created with *pip*
* Giving way to portability across systems

Using *virtualenv* is the recommended way for working with Python projects, regardless of how many you might be busy with. It is very easy to use and an excellent tool to have at your disposable. It truly does wonders when coupled with *pip.*

**How to Install ?**

In order to install *virtualenv*, we are going to call in *pip* for help. We will install it as a globally available package for the Python interpreter to run.

There are two ways to obtain the application. The version you will be able to get depends on which one you choose.

The simplest method is using *pip* to search, download and install. This might not provide you the latest stable version.

**sudo pip install virtualenv**

**How to use ?**

Using this tool consists of getting it to create a folder, containing the Python interpreter and a copy of *pip*. Afterwards, in order to work with it, we need to either specify the location of that interpreter or activate it.

All the applications you install using the interpreter inside the virtual environment will be places within that location.

When you use *pip* to create a list of them, only the ones inside the folder will be compiled into a file.

**Creating an environment using the same interpreter used to run it:**

# Example: virtualenv [folder (env.) name]  
 # Let's create an environment called \*my\_app\*  
 **virtualenv my\_app**

**Creating an environment with a custom Python interpreter:**

# Example: virtualenv --python=[loc/to/python/] [env. name]  
 **virtualenv --python=/opt/python-3.3/bin/python my\_app**

**Activating a *virtual environment***

# Example: source [env. name]/bin/activate  
 # Let's activate the Python environment we just created  
 source my\_app/bin/activate

### Working with a *virtual environment* without activating : -

For various reasons, you might choose not to activate the environment before using it. This brings more flexibility to commands you run, however, you need to make sure to target the correct interpreter each and every time.

# Example: [env. name]/bin/python [arguments]  
 my\_app/bin/python python\_script.py

**Using the *pip* installation inside the environment without activation**

# Example: [env. name]/bin/pip [command] [arguements]  
# Let's install requests library without activating the env.  
 my\_app/bin/pip install requests

**Deactivating a *virtual environment*:**

# Example: deactivate  
# Let's deactivate the environment from earlier  
 deactivate

For more details about virtual environment and PIP refer : -

<https://www.digitalocean.com/community/tutorials/common-python-tools-using-virtualenv-installing-with-pip-and-managing-packages>

***Fabric***

**What is Fabric ?**

Fabric is a Python library (i.e. a tool to build *on*) used for interacting with SSH and computer systems [easily] to automate a wide range of tasks, varying from application deployment to general system administration.

**Why to Use ?**

Albeit being Python-based, it does not mean that it is used strictly for working with other Python applications or tools. In fact, Fabric is there for you to achieve just about anything regardless of a specific language or a system. As long as the very basic requirements are met, you can take advantage of this excellent library.

Fabric scripts are basic Python files. They are run using the fab tool that is shipped with with Fabric. All this does is include (i.e. import ..) your script (i.e. instructions to perform) and execute the provided procedure.

### System/Server Administration -

One of the key areas for using Fabric is automating the everyday tasks of system (and server) administration. These jobs include pretty much everything that relates to:

* Building a server;
* Its maintenance, and;
* Monitoring.

### Application Deployment -

Deploying an application (regardless of it being a web site, an API, or a server) usually means setting up a system from scratch (or from a snapshot taken in time), preparing it by updating everything, downloading dependencies, setting up the file structure and permissions, followed by finally uploading your codebase - or downloading it using a SCM such as Git.

During the development process, you are also likely to have commands that need to be routinely executed (ex: right before entering a deployment cycle).

Being able to script these tasks (both local and remote) in a logically organized and -- most importantly -- programmable manner proves to be invaluable shortly after you realize how much time is being wasted repeating the same steps constantly, rendering everything error-prone during the process.

This is exactly when Fabric comes to your aid in the form of a Python file that will know *what* to do and *where* to do it.

**How to Install ?**

An easy and cohesive way of installing Fabric is by using the default operating system package manager aptitude.

In order to install Fabric using aptitude, run the following:

sudo aptitude install fabric  
  
# Alternatively, you can also use \*pip\*:  
# pip install fabric

Out of the box, any Python command (or procedure) and module can be utilised through Fabric - given that Fabric is indeed a Python library.

What Fabric really brings to the table is its extensive and excellent integration with SSH that allows streamlining everything using simple scripts (i.e. fabfile.py).

In this section, you can find a selection of tools (e.g. functions) that come with Fabric which can be used to interact with environments where commands you specify are executed.

## Fabric's Features and Integration with SSH

Out of the box, any Python command (or procedure) and module can be utilised through Fabric - given that Fabric is indeed a Python library.

What Fabric really brings to the table is its extensive and excellent integration with SSH that allows streamlining everything using simple scripts (i.e. fabfile.py).

In this section, you can find a selection of tools (e.g. functions) that come with Fabric which can be used to interact with environments where commands you specify are executed.

Note: You can see and learn more about Fabric's operations by visiting its documentation [on the subject](http://docs.fabfile.org/en/1.4.0/api/core/operations.html).

### run (fabric.operations.run)

Fabric's run procedure is used for executing a shell command on one or more remote hosts.

* The output results of *run* can be captured using a variable.
* If command succeeded or failed can be checked using .failed and .succeeded.

**Usage examples:**

**# Create a directory (i.e. folder)  
run("mkdir /tmp/trunk/")  
  
# Uptime  
run("uptime")  
  
# Hostname  
run("hostname")  
  
# Capture the output of "ls" command  
result = run("ls -l /var/www")  
  
# Check if command failed  
result.failed**

### sudo (fabric.operations.sudo)

Along with run, the most widely used Fabric command is probably sudo. It allows the execution of a given set of commands and arguments with sudo (i.e. *superuser*) privileges on the remote host.

If sudo command is used with an explicitly specified user, the execution will happen not as root but another (i.e. UID 1010).

**Usage examples:**

**# Create a directory  
sudo("mkdir /var/www")  
  
# Create a directory as another user  
sudo("mkdir /var/www/web-app-one", user="web-admin")  
  
# Return the output  
result = sudo("ls -l /var/www")**

### local (fabric.operations.local)

As we have mentioned in our introduction, a single Fabric script (fabfile) can be used to perform actions both on the local machine and remote system(s). For this purpose, Fabric provides the local operative to run commands locally.

Unlike run or sudo, however, interacting with the output of local the same way is not possible. Either output can be captured or printed -- the switch can be set with capture argument.

Local helpers such as the lcd context manager (which is used for setting the local working directory) are *honoured* with local, the same way run (or sudo) honours the cd context manager.

**Usage examples:**

**# Create a source distribution tar archive (for a Python App.)  
local("python setup.py sdist --formats=gztar", capture=False)  
  
# Extract the contents of a tar archive  
local("tar xzvf /tmp/trunk/app.tar.gz")  
  
# Remove a file  
local("rm /tmp/trunk/app.tar.gz")**

### get (fabric.operations.get)

The get command exists to download (i.e. pull) file(s) from the remote system to the computer where the Fabric is being used. It is similar to how scp works and comes in handy when you need to download backups, logging data or some other server related items.

* You can specify the remote path with the remote\_path argument.
* You can specify the local - download - path with the local\_path argument.

**Usage examples:**

**# Download some logs  
get(remote\_path="/tmp/log\_extracts.tar.gz", local\_path="/logs/new\_log.tar.gz")  
  
# Download a database back-up  
get("/backup/db.gz", "./db.gz")**

### put (fabric.operations.put)

When you need to upload files, put command can be used very similarly to get. You can again access the results of command's execution with .failed or .succeeded.

* local\_path - set the local path.
* remote\_path - set the remote path.
* use\_sudo - upload the file to anywhere on the remote machine using a nifty trick: upload to a temporary location then move.
* mode - set the file mode (flags).
* mirror\_local - set the file flags (i.e. make executable) automatically by reading the local file's model.

**Usage examples:**

**# Upload a tar archive of an application  
put("/local/path/to/app.tar.gz", "/tmp/trunk/app.tar.gz")  
  
# Use the context manager `cd` instead of "remote\_path" arg.  
# This will upload app.tar.gz to /tmp/trunk/  
with cd("/tmp"):  
 put("local/path/to/app.tar.gz", "trunk")  
  
# Upload a file and set the exact mode desired  
upload = put("requirements.txt", "requirements.txt", mode=664)  
  
# Verify the upload  
upload.succeeded**

### prompt (fabric.operations.prompt)

When you find yourself in need of some extra flexibility working with Fabric, prompt will come to your rescue. This command does exactly what its name suggests and asks the user (i.e. one that is running the script) to input a certain data to use during the successive execution.

If you are using a single file to manage with multiple applications, for example, you can use prompt to set one to perform the actions.

Before starting with anything, prompt can also be used to query the port number to use.

**Usage examples:**

**# Prompt the user  
port\_number = prompt("Which port would you like to use?")  
  
# Prompt the user with defaults and validation  
port\_number = prompt("Which port?", default=42, validate=int)**

### reboot (fabric.operations.reboot)

The reboot command is also self explanatory: it is used to reboot the remote system. By default, it waits two minutes (i.e. 120 seconds -> wait=120) before doing its job.

**Usage examples:**

**# Reboot the remote system  
reboot()  
  
# Reboot after 30 seconds  
reboot(wait=30)**

for More Details Refer :-

<https://www.digitalocean.com/community/tutorials/how-to-use-fabric-to-automate-administration-tasks-and-deployments>

***Django***

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.Django is written in 100% pure Python code, so you’ll need to install Python on your system. Django requires Python 2.3 or higher.

**Installing Django** :

You can install Django on your system(Linux) by the below command :-

Django **sudo pip install Django==1.6(version)**

To verify that Django can be seen by Python, type python from your shell. Then at the Python prompt, try to import Django like :-

**>>> import django**

**>>> print(django.get\_version())**

**1.6**

**Starting a Project in Django :**

A project is a collection of settings for an instance of Django, including database configuration,Django-specific options, and application-specific settings.

If this is your first time using Django, you’ll have to take care of some initial setup. Create a new directory to start working in, perhaps something like /home/username/djcode/, and change into that directory.

**CheckList before Starting a new Project :**

* **django-admin.py** should be on your system path if you installed Django via its **setup.py** utility. If you checked out from Subversion, you can find it in djtrunk/django/bin. Since you’ll be using django-admin.py often, consider adding it to your path. On Unix, you can do so by symlinking from /usr/local/bin, using a command

**sudo ln -s /path/to/django/bin/django-admin.py /usr/local/bin/django-admin.py**.

* **On Windows,** you’ll need to update your PATH environment variable.

Refer the below links for more details : -

*Downloading and installing Python and Django on Windows*

<http://www.youtube.com/watch?v=rIVwVOpwpsA>

*Configuring : -*

<http://www.youtube.com/watch?v=NqNNuXwi0eI>

* **Create a new project** by running the command from the command line, cd into a directory where you'd like to store your code

**django-admin.py startproject mysite**

This will create a mysite directory(**Your Project**) in your current directory. Let’s look at what startproject created:

**mysite/**

**\_\_init\_\_.py**

**manage.py**

**settings.py**

**urls.py**

***These files are as follows : -***

1. \_\_init\_\_.py: A file required for Python treat the directory as a package (i.e., a group of modules)
2. manage.py: A command-line utility that lets you interact with this Django project in various ways
3. settings.py: Settings/configuration for this Django project
4. urls.py: The URL declarations for this Django project; a “table of contents” of your Django-powered site.

* **Creating an App inside your created Project :-**   
  The Django app ecosystem provides a large number of apps that can be added to your project. Many are mature and used in many existing projects and sites, while others are under active early-stage development. Websites such as[**Django Packages**](http://www.djangopackages.com/) provide an interface for discovering and comparing all the apps that can plugged in to your Django project. You will find that some can be used with very little effort on your part, and some will take more effort to integrate.

You can create an app inside the Project using below command

**python manage.py startapp myfristapp**

**Development Server in Django :**

Django includes a built-in, lightweight Web server you can use while developing your site. We’ve included this server so you can develop your site rapidly, without having to deal with configuring your production Web server (e.g., Apache) until you’re ready for production. This development server watches your code for changes and automatically reloads, helping you make many rapid changes to your project without needing to restart anything.

Change into the mysite directory, if you haven’t already, and run the command

**python manage.py runserver**

You’ll see something like this:

**Validating models...**

**0 errors found.**

**Django version 1.0, using settings 'mysite.settings'**

**Development server is running at http://127.0.0.1:8000/**

**Quit the server with CONTROL-C**.

**Changing the Host or the Port :**

By default, the runserver command starts the development server on por**t 8000,** listening only for local connections. If you want to change the server’s port, pass it as a command-line argument:

**python manage.py runserver 8080**

You can also change the **IP address** that the server listens on. This is especially helpful if you’d like to share a development site with other developers. The following:

**python manage.py runserver 0.0.0.0:8080**

will make Django listen on any network interface, thus allowing other computers to connect to the development server.

When you, visit http://127.0.0.1:8000/ with your Web browser. You’ll see a

**“Welcome to Django” page shaded a pleasant pastel blue. It worked**!

References:-

[www.djangoproject.com](http://www.djangoproject.com)

[www.djangobook.com](http://www.djangoproject.com)

**Steps for Creating Models :**

1. Define your model in "models.py" of your app

2. In settings.py of your project add the the app name under "INSTALLED\_APPS"

3. Run Command "python manage.py sql <appname>" -- Gives you the sql query which Django going to fire for creating the tables in your DB(This will not creates the table in database)

4. Run "**python manage.py syncdb**"-This will Create the Tables in your Database.

5. If you want to see the table from the admin panel of Django then you need to Register it in "admin.py" of your app

By adding-  **from projectname.models import <modelname>**

**admin.site.register(modelname)**

6. Creating superusers

**manage.py syncdb**

prompts you to create a superuser the first time you run it with 'django.contrib.auth' in your INSTALLED\_APPS. If you need to create a superuser at a later date, you can use a command line utility:

**$ python manage.py createsuperuser --username=joe --email=joe@example.com**

You will be prompted for a password. After you enter one, the user will be created immediately. If you leave off the --username or the --email options, it will prompt you for those values.

**Changing passwords**

Django does not store raw (clear text) passwords on the user model, but only a hash (see documentation of how passwords are managed for full details). Because of this, do not attempt to manipulate the password attribute of the user directly. This is why a helper function is used when creating a user.

**To change a user’s password, you have several options**

manage.py changepassword \*username\* offers a method of changing a User’s password from the command line. It prompts you to change the password of a given user which you must enter twice. If they both match, the new password will be changed immediately. If you do not supply a user, the command will attempt to change the password whose username matches the current system user.

You can also change a password programmatically, using set\_password():

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

**>>> u = User.objects.get(username\_\_exact='john')**

**>>> u.set\_password('new password')**

**>>> u.save()**

**Steps for Creating the Views :-**

1. Let’s write the first view. Open the file views.py in your module and put the following Python code in it:

**from django.http import HttpResponse**

**def index(request):**

**return HttpResponse("Hello, world. You're at the poll index.”)**

2. 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 App directory, create a file called urls.py. Your app directory should now look like:

**module/**

**\_\_init\_\_.py**

**admin.py**

**models.py**

**tests.py**

**urls.py**

**views.py**

In the module/urls.py file include the following code:

**from django.conf.urls import patterns, url**

**from <modulename> import views**

**urlpatterns = patterns('',**

**url(r'^$', views.index, name='index')**

**)**

3. The next step is to point the root URLconf at the apps.urls module. In Project/urls.py insert an include(), leaving you with:

**from django.conf.urls import patterns, include, url**

**from django.contrib import admin**

**admin.autodiscover()**

**urlpatterns = patterns('',**

**url(r'^polls/', include('<modulname>.urls')),**

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

**)**

You have now wired an **index** view into the **URLconf**. Go t**o http://localhost:8000/modulename**/ in your browser, and you should see the text “Hello, world. You’re at the modulename index.”, which you defined in th**e ind**ex view.

***Django Templates***

**Django templates : -**

A Django template is a string of text that is intended to separate the presentation of a document from its data.

A template defines placeholders and various bits of basic logic like “template tags” that regulate how the document should be displayed. Usually, templates are used for producing HTML, but Django templates are equally capable of generating any text-based format.

Let’s dive in with a simple example template. This template describes an HTML page that thanks a person for placing an order with a company. Think of it as a form letter:

**<html>**

**<head><title>Ordering notice</title></head>**

**<body>**

**<p>Dear {{ person\_name }},</p>**

**<p>Thanks for placing an order from {{ company }}. It's scheduled to**

**ship on {{ ship\_date|date:"F j, Y" }}.</p>**

**<p>Here are the items you've ordered:</p>**

**<ul>**

**{% for item in item\_list %}**

**<li>{{ item }}</li>**

**{% endfor %}**

**</ul>**

**{% if ordered\_warranty %}**

**<p>Your warranty information will be included in the packaging.</p>**

**{% endif %}**

**<p>Sincerely,<br />{{ company }}</p>**

**</body>**

**</html>**

* Any text surrounded by a pair of braces (e.g., {{ person\_name }}) is a ***variable***. This means “insert the value of the variable with the given name.”
* Any text that’s surrounded by curly braces and percent signs

{% if ordered\_warranty %}) is a ***template tag***.

* This example template contains two tags:

the {% for item in item\_list %} tag (a for tag) and

the {% if ordered\_warranty %} tag (an if tag).

* A **for tag** acts as a simple loop construct, letting you loop over each item in a sequence. An **if tag,** as you may expect, acts as a logical “if” statement. In this particular case, the tag checks whether the value of the ordered\_warranty variable evaluates to True. If it does, the template system will display everything between the

**{% if ordered\_warranty %}**

**and**

**{% endif %}.**

If not, the template system won’t display it. The template system also supports {% else %} and other various logic statements.

* Finally, the second paragraph of this template has an example of a *filter*, with which you can alter the display of a variable. In this example,

**{{ ship\_date|date:"F j, Y" }}**, we’re passing the ship\_date variable to the date filter, giving the date filter the argument "F j, Y".

* The date filter formats dates in a given format, as specified by that argument. Filters are attached using a pipe character (|), as a reference to Unix pipes.

**Rendering a Template**

Once you have a Template object, you can pass it data by giving it a ***context***. A context is simply a set of variables and their associated values. A template uses this to populate its variable tags and evaluate its block tags.

A context is represented in Django by the Context class, which lives in the django.template module.

Its constructor takes one optional argument: a dictionary mapping variable names to variable values. Call the Template object’s **render()** method with the context to “fill” the template:

>>> from django.template import Context, Template

>>> t = Template("My name is {{ name }}.")

>>> c = Context({"name": "Stephane"})

>>> t.render(c)

'My name is Stephane.'

**Dictionaries and Contexts**

A Python dictionary is a mapping between known keys and variable values. A Context is similar to a

dictionary, but a Context provides additional functionality.

Variable names must begin with a letter (A-Z or a-z) and may contain digits, underscores, and dots. (Dots are a special case we’ll get to in a moment.) Variable names are case sensitive.

Here’s an example of template compilation and rendering, using the sample template from the beginning of this chapter:

**>>> from django.template import Template, Context**

**>>> raw\_template = """<p>Dear {{ person\_name }},</p>**

**...**

**... <p>Thanks for ordering {{ product }} from {{ company }}. It's scheduled**

**... to ship on {{ ship\_date|date:"F j, Y" }}.</p>**

**...**

**... {% if ordered\_warranty %}**

**... <p>Your warranty information will be included in the packaging.</p>**

**... {% endif %}**

**...**

**... <p>Sincerely,<br />{{ company }}</p>"""**

**>>> t = Template(raw\_template)**

**>>> import datetime**

**>>> c = Context({'person\_name': 'John Smith',**

**... 'product': 'Super Lawn Mower',**

**... 'company': 'Outdoor Equipment',**

**... 'ship\_date': datetime.date(2009, 4, 2),**

**... 'ordered\_warranty': True})**

**>>> t.render(c)**

**"<p>Dear John Smith,</p>\n\n<p>Thanks for ordering Super Lawn Mower from**

**Outdoor Equipment. It's scheduled \nto ship on April 2, 2009.</p>\n\n\n**

**<p>Your warranty information will be included in the packaging.</p>\n\n\n**

**<p>Sincerely,<br />Outdoor Equipment</p>"**

**Let’s step through this code one statement at a time:**

1. First, we import the classes Template and Context, which both live in the module django.template.

2. We save the raw text of our template into the variable raw\_template. Note that we use triple quote marks to designate the string, because it wraps over multiple lines; in Python codde, strings designated with single quote marks cannot be wrapped over multiple lines.

3. Next, we create a template object, t, by passing raw\_template to the Template class constructor.

4. We import the datetime module from Python’s standard library, because we’ll need it in the following statement.Then, we create a Context object, c. The Context constructor takes a Python dictionary, which maps variable names to values. Here, for example, we specify that the person\_name is 'John Smith', product is 'Super Lawn Mower', and so forth.

5. Finally, we call the render() method on our template object, passing it the context. This returns the rendered template—that is, it replaces template variables with the actual values of the variables, and it executes any block tags.

Note that the warranty paragraph was displayed because the ordered\_warranty variable evaluated to True. Also note the date, April 2, 2009, which is displayed according to the format string 'F j, Y'.If you’re new to Python, you may wonder why this output includes newline characters ('\n') rather than displaying the line breaks. That’s happening because of a subtlety in the Python interactive interpreter: the call to t.render(c) returns a string, and by default the interactive interpreter displays the *representation* of the string, rather than the printed value of the string. If you want to see the string with line breaks displayed as true line breaks rather than '\n' characters, use the print statement: print t.render(c).

Those are the fundamentals of using the Django template system: just write a template, create a Template object, create a Context, and call the render() method.

**Basic Template Tags and Filters :**

**Tags:-**

**1. If/else : -**

* The {% if %} tag evaluates a variable, and if that variable is “true” (i.e., it exists, is not empty, and is not a false Boolean value), the system will display everything between {% if %} and {% endif %}
* Python “Truthiness”

In Python, the empty list ([]), tuple (()), dictionary ({}), string (''), zero (0), and the special object None are False in a Boolean context. Everything else is True.

* The {% if %} tag accepts and, or, or not for testing multiple variables, or to negate a given variable. For example :

**{% if athlete\_list and coach\_list %}**

**Both athletes and coaches are available.**

**{% endif %}**

**{% if not athlete\_list %}**

**There are no athletes.**

**{% endif %}**

**{% if athlete\_list or coach\_list %}**

**There are some athletes or some coaches.**

**{% endif %}**

**{% if not athlete\_list or coach\_list %}**

**There are no athletes or there are some coaches. (OK, so**

**writing English translations of Boolean logic sounds**

**stupid; it's not our fault.)**

**{% endif %}**

**{% if athlete\_list and not coach\_list %}**

**There are some athletes and absolutely no coaches.**

**{% endif %}**

* {% if %} tags don’t allow and and or clauses within the same tag, because the order of logic would be ambiguous. For example, this is invalid:

**{% if athlete\_list and coach\_list or cheerleader\_list %}**

The use of parentheses for controlling order of operations is not supported. If you find yourself needing parentheses, consider performing logic in the view code in order to simplify the templates. Even so, if you need to combine and and or to do advanced logic, just use nested {% if %} tags, for example:

**{% if athlete\_list %}**

**{% if coach\_list or cheerleader\_list %}**

We have athletes, and either coaches or cheerleaders!

**{% endif %}**

**{% endif %}**

* Multiple uses of the same logical operator are fine, but you can’t combine different operators. For example this is valid:

**{% if athlete\_list or coach\_list or parent\_list or teacher\_list %}**

* There is no {% elif %} tag. Use nested {% if %} tags to accomplish the same thing:

**{% if athlete\_list %}**

**<p>Here are the athletes: {{ athlete\_list }}.</p>**

**{% else %}**

**<p>No athletes are available.</p>**

**{% if coach\_list %}**

**<p>Here are the coaches: {{ coach\_list }}.</p>**

**{% endif %}**

**{% endif %}**

Make sure to close each **{% if %}** with an **{% endif %}**. Otherwise, Django will throw a TemplateSyntaxError.

2. **for:-**

* The {% for %} tag allows you to loop over each item in a sequence. As in Python’s for statement, the syntax is for X in Y, where Y is the sequence to loop over and X is the name of the variable to use for a particular cycle of the loop. Each time through the loop, the template system will render everything between

{% for %} and {% endfor %}.

For example, you could use the following to display a list of athletes given a variable athlete\_list:

**<ul>**

**{% for athlete in athlete\_list %}**

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

**{% endfor %}**

**</ul>**

* There is no support for “breaking out” of a loop before the loop is finished. If you want to accomplish this, change the variable you’re looping over so that it includes only the values you want to loop over. Similarly, there is no support for a “continue” statement that would instruct the loop processor to return immediately to the front of the loop.
* The {% for %} tag sets a magic forloop template variable within the loop. This variable has a few attributes that give you information about the progress of the loop:

1.  **forloop.counter** is always set to an integer representing the number of times the loop has been entered. This is one-indexed, so the first time through the loop, forloop.counter will be set to 1.

2. **forloop.counter0**is like forloop.counter, except it’s zero-indexed. Its value will be set to0 the first time through the loop.

3. forloop.revcounter is always set to an integer representing the number of remaining items in the loop. The first time through the loop, forloop.revcounter will be set to the total number of items in the sequence you’re traversing. The last time through the loop, forloop.revcounter will be set to 1.

4. **forloop.revcounter0** is like forloop.revcounter, except it’s zero-indexed. The first time through the loop, forloop.revcounter0 will be set to the number of elements in the sequence minus 1. The last time through the loop, it will be set to 0.

5. forloop.first is a Boolean value set to True if this is the first time through the loop. This is convenient for special casing:

**{% for object in objects %}**

**{% if forloop.first %}<li class="first">{% else %}<li>{% endif %}**

**{{ object }}**

**</li>**

**{% endfor %}**

**DataBase Drivers Required**

**Connection with PostgreSQL :**

Several libraries exist for connecting to the PostgreSQL database from the Python language.We will use the psycopg2 module. It is a PostgreSQL database adapter for the Python programming language. According to the module documentation it is currently the most popular Python module for the PostgreSQL database. It is mostly implemented in C as a libpq wrapper.

## About PostgreSQL database

PostgreSQL is a powerful, open source object-relational database system. It is a multi-user database management system. It runs on multiple platforms including Linux, FreeBSD, Solaris, Microsoft Windows and Mac OS X. PostgreSQL is developed by the PostgreSQL Global Development Group.

To Start work with, we must have Python language, PostgreSQL database and psycopg2 language binding installed on our system.

$ sudo apt-get install postgresql

On an Ubuntu based system we can install the PostgreSQL database using the above command.

$ sudo update-rc.d -f postgresql remove  
 Removing any system startup links for /etc/init.d/postgresql ...  
 /etc/rc0.d/K21postgresql  
 /etc/rc1.d/K21postgresql  
 /etc/rc2.d/S19postgresql  
 /etc/rc3.d/S19postgresql  
 /etc/rc4.d/S19postgresql  
 /etc/rc5.d/S19postgresql  
 /etc/rc6.d/K21postgresql

If we install the PostgreSQL database from packages, it is automatically added to the start up scripts of the operating system. If we are only learning to work with the database, it is unnecessary to start the database each time we boot the system. The above command removes any system startup links for the PostgreSQL database.

$ /etc/init.d/postgresql status  
Running clusters: 9.1/main  
  
$ service postgresql status  
Running clusters: 9.1/main

We check if the PostgreSQL server is running. If not, we need to start the server.

$ sudo service postgresql start  
 \* Starting PostgreSQL 9.1 database server [ OK ]

On Ubuntu Linux we can start the server with the service postgresql start command.

$ sudo service postgresql stop  
[sudo] password for janbodnar:   
 \* Stopping PostgreSQL 9.1 database server [ OK ]

We use the service postgresql stop command to stop the PostgreSQL server.

$ sudo apt-get install python-psycopg2

Here we install the psycopg2 module on a Ubuntu system.

$ sudo -u postgres createuser janbodnar  
Shall the new role be a superuser? (y/n) n  
Shall the new role be allowed to create databases? (y/n) y  
Shall the new role be allowed to create more new roles? (y/n) n

We create a new role in the PostgreSQL system. We allow it to have ability to create new databases. A *role* is a user in a database world. Roles are separate from operating system users. We have created a new user without the -W option, e.g. we have not specified a password. This enables us to connect to a database with this user without password authentication. Note that this works only on localhost.

$ sudo -u postgres createdb testdb -O janbodnar

The createdb command creates a new PostgreSQL database with the owner janbodnar.

In the first code example, we will get the version of the PostgreSQL database.

**#!/usr/bin/python  
# -\*- coding: utf-8 -\*-  
  
import psycopg2  
import sys  
con = None  
try:  
 con = psycopg2.connect(database='testdb', user='janbodnar')   
 cur = con.cursor()  
 cur.execute('SELECT version()')   
 ver = cur.fetchone()  
 print ver**

**except psycopg2.DatabaseError, e:  
 print 'Error %s' % e   
 sys.exit(1)  
finally:  
   
 if con:  
 con.close()**

**Explanation Of Above Code Snippet :**

In the above Python script we connect to the previously created testdb database. We execute an SQL statement which returns the version of the PostgreSQL database.

**import psycopg2**

The psycopg2 is a Python module which is used to work with the PostgreSQL database.

**con = None**

We initialize the con variable to None. In case we could not create a connection to the database (for example the disk is full), we would not have a connection variable defined. This would lead to an error in the finally clause.

**con = psycopg2.connect(database='testdb', user='janbodnar')**

The connect() method creates a new database session and returns a connection object. The user was created without a password. On localhost, we can omit the password option. Otherwise, it must be specified.

**cur = con.cursor()  
cur.execute('SELECT version()')**

From the connection, we get the cursor object. The cursor is used to traverse the records from the result set. We call the execute() method of the cursor and execute the SQL statement.

**er = cur.fetchone()**

We fetch the data. Since we retrieve only one record, we call the fetchone() method.

**print ver**

We print the data that we have retrieved to the console.

**except psycopg2.DatabaseError, e:  
 print 'Error %s' % e   
 sys.exit(1)**

In case of an exception, we print an error message and exit the script with an error code 1.

**finally:  
   
 if con:  
 con.close())**

In the final step, we release the resources.

**$ ./version2.py**('PostgreSQL 9.3.5 on i686-pc-linux-gnu, compiled by gcc   
 (Ubuntu 4.8.2-19ubuntu1) 4.8.2, 32-bit',)

Running the version.py script.

For More Details Refer : - <http://zetcode.com/db/postgresqlpythontutorial/>

**Using Mysql with Python :**

This is a Python programming tutorial for the MySQL database. It covers the basics of MySQL programming with Python. It uses the MySQLdb module. The examples were created and tested on Ubuntu Linux.

## Before we start

We need to install several packages to execute the examples below

If you do not already have MySQL installed, we must install it.

**$ sudo apt-get install mysql-server**

This command installs the MySQL server and various other packages. While installing the package, we are prompted to enter a password for the MySQL root account.

**$ apt-cache search MySQLdb  
python-mysqldb - A Python interface to MySQL**  
**python-mysqldb-dbg** - A Python interface to MySQL (debug extension)  
**bibus** - bibliographic database  
**eikazo** - graphical frontend for SANE designed for mass-scanning

We don not know the package name for the MySQLdb module. We use the **apt-cache** command to figure it out.

**$ sudo apt-get install python-mysqldb**

Here we install the Python interface to the MySQL database. Both \_mysql and MySQL modules.

Next, we are going to create a new database user and a new database. We use the mysql client.

**$ mysql -u root -p  
Enter password:**   
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 30  
Server version: 5.0.67-0ubuntu6 (Ubuntu)  
  
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.  
  
mysql> SHOW DATABASES;  
+--------------------+  
| Database |  
+--------------------+  
| information\_schema |   
| mysql |   
+--------------------+  
2 rows in set (0.00 sec)

We connect to the database using the root account. We show all available databases with the SHOW DATABASES statement.

mysql> CREATE DATABASE testdb;  
Query OK, 1 row affected (0.02 sec)

We create a new testdb database. We will use this database throughout the tutorial.

mysql> CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'test623';  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> USE testdb;  
Database changed  
  
mysql> GRANT ALL ON testdb.\* TO 'testuser'@'localhost';  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> quit;  
Bye

We create a new database user. We grant all privileges to this user for all tables of the testdbdatabase.

## \_mysql module

The \_mysql module implements the MySQL C API directly. It is not compatible with the Python DB API interface. Generally, the programmers prefer the object oriented MySQLdb module. We will concern ourself with the latter module. Here we present only one small example with the \_mysqlmodule.

**#!/usr/bin/python  
# -\*- coding: utf-8 -\*-  
  
import \_mysql  
import sys  
  
try:  
 con = \_mysql.connect('localhost', 'testuser', 'test623', 'testdb')  
   
 con.query("SELECT VERSION()")  
 result = con.use\_result()  
   
 print "MySQL version: %s" % \  
 result.fetch\_row()[0]  
   
except \_mysql.Error, e:  
   
 print "Error %d: %s" % (e.args[0], e.args[1])  
 sys.exit(1)  
  
finally:  
   
 if con:  
 con.close()**

The example will get and print the version of the MySQL database. For this, we use the SELECT VERSION() SQL statement.

## MySQLdb module

MySQLdb is a thin Python wrapper around \_mysql. It is compatible with the Python DB API, which makes the code more portable. Using this model is the preferred way of working with the MySQL.

## First example

In the first example, we will get the version of the MySQL database.

**#!/usr/bin/python  
# -\*- coding: utf-8 -\*-  
  
import MySQLdb as mdb  
import sys  
  
try:  
 con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');  
  
 cur = con.cursor()  
 cur.execute("SELECT VERSION()")  
  
 ver = cur.fetchone()  
   
 print "Database version : %s " % ver  
   
except mdb.Error, e:  
   
 print "Error %d: %s" % (e.args[0],e.args[1])  
 sys.exit(1)  
   
finally:   
   
 if con:   
 con.close()**

In this script, we connect to the testdb database and execute the SELECT VERSION() statement. This will return the current version of the MySQL database. We print it to the console.

**import MySQLdb as mdb**

We import the MySQLdb module.

**con = mdb.connect('localhost', 'testuser',   
 'test623', 'testdb');**

We connect to the database. The connect() method has four parameters. The first parameter is the host, where the MySQL database is located. In our case it is a localhost, e.g. our computer. The second parameter is the database user name. It is followed by the user's account password. The final parameter is the database name.

**cur = con.cursor()  
cur.execute("SELECT VERSION()")**

From the connection, we get the cursor object. The cursor is used to traverse the records from the result set. We call the execute() method of the cursor and execute the SQL statement.

**ver = cur.fetchone()**

We fetch the data. Since we retrieve only one record, we call the fetchone() method.

print "Database version : %s " % ver

We print the data that we have retrieved to the console.

except mdb.Error, e:  
   
 **print "Error %d: %s" % (e.args[0],e.args[1])  
 sys.exit(1)**

We check for errors. This is important, since working with databases is error prone.

**finally:   
   
 if con:   
 con.close()**

In the final step, we release the resources.

$ ./version.py  
Database version : 5.5.9

The output might look like the above.

For more details refer : - <http://zetcode.com/db/postgresqlpythontutorial/>

***South for Django Application***

**Introduction to South :**

Migrations are a way of changing your database schema from one version into another.

South brings migrations to Django applications its main objectives are to provide a simple, stable and database-independent migration layer to prevent all the conflicting schema changes for your Django applications.

It is the most popular schema migration tool for Django.

**Key** **Features** **:**

Database Independent

Automatic migration creation

**Database supported by south :**

* PostgreSQL
* MySQL
* SQLite
* Microsoft SQL Server
* Oracle
* Firebird (beta support)

**Database Specific Issues :**

* PostgreSQL supports all of the South features.
* MySQL and Oracle don’t have transaction support for schema modification
* SQLite doesn’t natively support much schema altering at all, but South has workarounds to allow deletion/altering of columns.
* Firebird almost all features are supported but need more tests. Renames table is not supported by firebird, this involve recreate all related objects.

**Installation :**

If you have pip or easy\_install available on your system, just type:-

**pip install South  
or  
easy\_install South**

If you’ve already got an old version of South, and want to upgrade, use:-

**pip install -U South  
or  
easy\_install -U South**

**Using South with a Django Project :**

For using the South with Django you need know about

How to create a project in Django ?

How to Start an App in Django ?

Django project structure ?

Django Project Structure:

**Create a new project** by running the command from the command line, cd into a directory where you'd like to store your code

**django-admin.py startproject mysite**

Project structure created by Django

**mysite/**

**\_\_init\_\_.py**

**manage.py**

**settings.py**

**urls.py**

You can create an app inside the Project using below command

**python manage.py startapp myfristapp**

**Configuring South with Django :**

Edit your settings.py and add 'south' to under the INSTALLED\_APPS.

**INSTALLED\_APPS = (**

**'django.contrib.admin',**

**'django.contrib.auth',**

**'django.contrib.contenttypes',**

**'django.contrib.sessions',**

**'django.contrib.messages',**

**'django.contrib.staticfiles',**

**'south'**

**)**

Once South is added in, you’ll need to run python manage.py syncdb to make the South migration-tracking tables.

**Creating migrations using south** **:-**

* **Creating Models :**

Each model is a Python class that subclasses [django.db.models.Model](https://docs.djangoproject.com/en/dev/ref/models/instances/#django.db.models.Model).

Each attribute of the model represents a database field

**from django.db import models**

**class Knight(models.Model):  
 name = models.CharField(max\_length=100)  
 of\_the\_round\_table = models.BooleanField()**

After Defineing Model run **python manage.py syncdb**

**syncdb -** Synchronizes the database state with the current set of models and migrations.

* **Creating Migrations :**

1. South has several ways of creating migrations; some are automatic, some are manual
2. There are two automatic ways --auto and --initial.
3. --auto looks at the previous migration, works out what’s changed, and creates a migration which applies the differences.
4. --initial, which will create tables and indexes for all of the models in the app; it’s what you use first, much like syncdb, and --auto is then used afterwards for each change.

* **Creating First Migration :**

Run the Command from terminal

**python manage.py schemamigration southtut --initial**

After running this you will get the below message **Creating migrations directory at '/home/andrew/Programs/litret/southtut/migrations'...  
Creating \_\_init\_\_.py in '/home/andrew/Programs/litret/southtut/migrations'...  
 + Added model southtut.Knight  
Created 0001\_initial.py. You can now apply this migration with: ./manage.py migrate southtut**

**Migration Structure :**

1. A migration file consist of a class called Migration inside with at least a **forwards()** and **backwards()** method.
2. Also Consist of Models Dictionary which stores the Current state of the Database
3. migrations are loaded in ASCII sort order, they won’t be applied in the correct order if you call them 1\_first, 2\_second, ..., 10\_tenth.
4. If you want to use numerical migrations in this fashion you prefix the numbers with zeroes like so: 0001\_first, 0002\_second, 0010\_tenth.

**Creating Migration for Changes done in existing model using south :**-

* **Change the existing Model :**

from django.db import models

**class** **Knight**(models**.**Model):  
 name **=** models**.**CharField(max\_length**=**100)  
 of\_the\_round\_table **=** models**.**BooleanField()  
 dances\_whenever\_able **=** models**.**BooleanField()

* **Create Migration for Changes :**

Make the new migration, using the **–auto** feature:

**python manage.py schemamigration southtut --auto** + Added field dances\_whenever\_able on southtut.Knight  
**Created 0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able.py.**

* **Applying Created Migration :**

**$ ./manage.py migrate southtut**  
Running migrations for southtut:  
 - Migrating forwards to 0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able.  
 > southtut:0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able  
 - Loading initial data for southtut

**Listing Existing Migration of Application :-**

It can be very useful to know what migrations you currently have applied, and which ones are available.

**python manage.py migrate --list southtut** (\*) 0001\_initial  
 (\*) 0002\_auto\_\_add\_field\_knight\_dances\_whenever\_able  
 (\*) 0003\_auto\_\_add\_field\_knight\_shrubberies  
 () 0004\_auto\_\_add\_unique\_knight\_name(Not Applied)

**python manage.py migrate --list |grep -v "\*"**

**Exceptions in Python**

Exceptions breaks the normal execution flow of an application,to avoid that we need to handel exceptions properly by using the Exception handling mechanism.

***Exceptions*** *handling in python mainly contains the four* ***Clauses*** *:-*

1.try

2.except

3.else

4.finally

**The try clause works as follows :**

* First, the try clause (the statement(s) between the try and except keywords) is executed.If no exception occurs, the except clause is skipped and execution of the try statement is finished.
* If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
* If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.
* A try statement may have more than one except clause, to specify handlers for different exceptions.At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause.

**Example** : -  
**try:**

**f = open('myfile.txt')**

**s = f.readline()**

**i = int(s.strip())**

**except IOError as e:**

**print "I/O error({0}): {1}".format(e.errno, e.strerror)**

**except ValueError:**

**print "Could not convert data to an integer."**

**except:**

**print "Unexpected error:", sys.exc\_info()[0]**

**raise**

* The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

import sys.

* The try ... except statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

**For example : -**

**for arg in sys.argv[1:]:**

**try:**

**f = open(arg, 'r')**

**except IOError:**

**print 'cannot open', arg**

**else:**

**print arg, 'has', len(f.readlines()), 'lines'**

**f.close()**

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances.

For example : -

**try:**

**... raise KeyboardInterrupt**

**... finally:**

**... print 'Goodbye!'**

**...**

**Goodbye!**