Contents

[Introduction 1](#_Toc366615165)

[Prerequisites 1](#_Toc366615166)

[Setup Endpoints for Virtual Machine 2](#_Toc366615167)

[With Portal 2](#_Toc366615168)

[With CLI 4](#_Toc366615169)

[Initial System Maintenance 5](#_Toc366615170)

[Login to the VM 5](#_Toc366615171)

[Upgrade Existing Packages 6](#_Toc366615172)

[Installing Python and Tools 7](#_Toc366615173)

[Installing Python 7](#_Toc366615174)

[Installing Easy Install and Pip 8](#_Toc366615175)

[Installing IPython 8](#_Toc366615176)

[Installing Additional Libraries 9](#_Toc366615177)

[Configuring IPython Notebook 10](#_Toc366615178)

[Creating Profile 10](#_Toc366615179)

[Creating Credentials 11](#_Toc366615180)

[Configuring IPython Notebook 11](#_Toc366615181)

[Starting IPython Notebook 12](#_Toc366615182)

[Running IPython Notebook in Background 13](#_Toc366615183)

[Access Azure Storage with Python 14](#_Toc366615184)

[Accessing Azure Storage from IPython Notebook 16](#_Toc366615185)

[Create Container 16](#_Toc366615186)

[Upload a File to Blob 16](#_Toc366615187)

[Read Contents from Blob 16](#_Toc366615188)

[Clustering Example with Pandas and Scikit-learn 16](#_Toc366615189)

# Introduction

The purpose of this document is to show the attendee how to set up iPython on an Azure VM with Ubuntu 12.04 LTS.

## Prerequisites

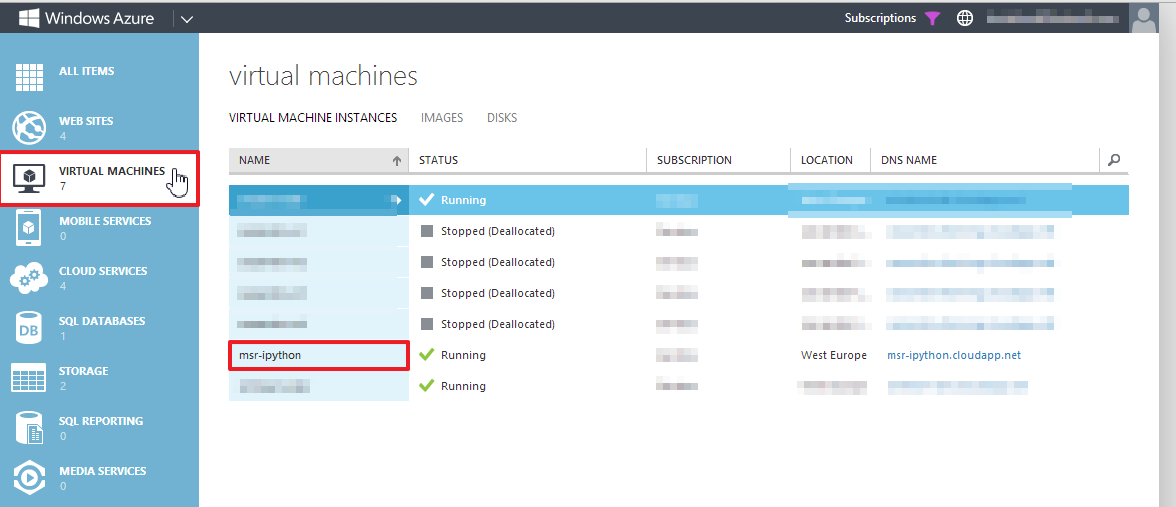
The attendee needs an Azure subscription and a VM running Ubuntu 12.04 LTS.

# Setup Endpoints for Virtual Machine

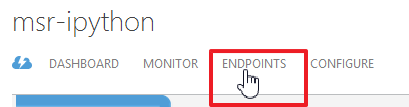
You will need to set up endpoints for your virtual machine so you can access the iPython notebook via web browser. You can do this in the web portal or from command line.

## With Portal

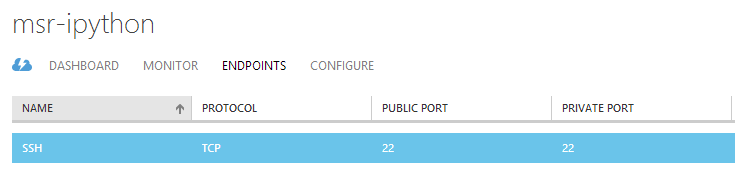
Go to the portal at <https://manage.windowsazure.com/>. Login with your credentials. On the left hand side, click on “Virtual Machines” and then on the virtual machine you want to use:



In the top bar, select “Endpoints”:



You will see that at the moment, only the SSH endpoint is defined:

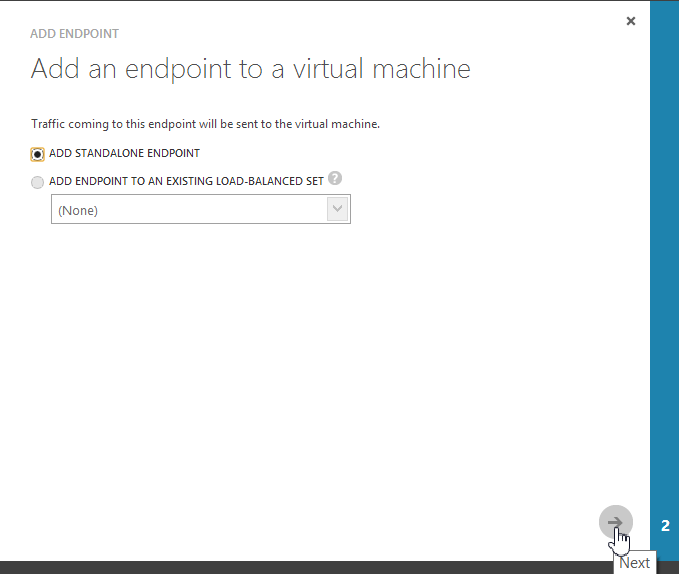


The IPython notebook runs on port 8888 in the default configuration. You can define any port as “public port”. A good choice might be port 443, the default port for HTTPS connections.

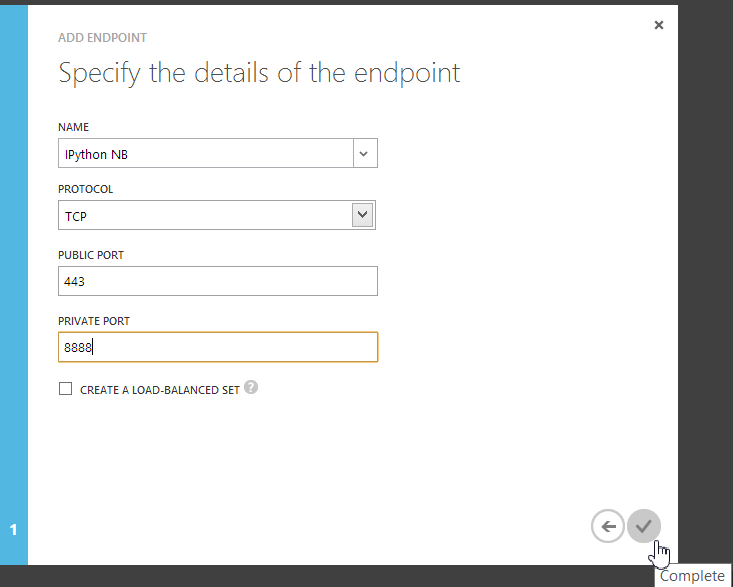
Click on “Add” in the bottom bar:



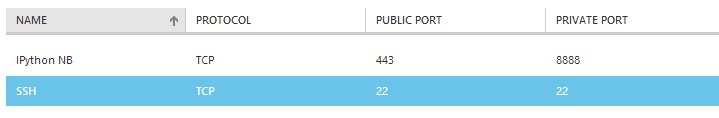
This will open a wizard. On the first screen, leave “Add standalone endpoint” selected. Then click on “Next”:

On the next screen, you can specify the endpoint details. Set

* Name: “IPython NB” (or any other endpoint name you like)
* Protocol: TCP
* Public Port: 443
* Private Port: 8888
* Keep “Create a load balanced set” unchecked

Then click on “Complete”:

The endpoint creation takes a short while. After that, you should see that it is configured:

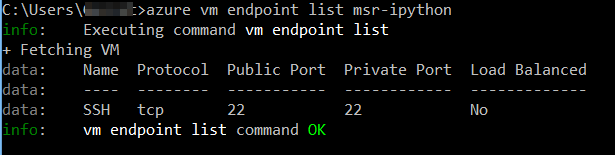


## With CLI

You can also configure the endpoint with the command line interface. To do this, open a command shell. To view the currently defined endpoints, type:

azure vm endpoint list <vm-name>

You should see that the SSH port is defined (public and private ports are 22):

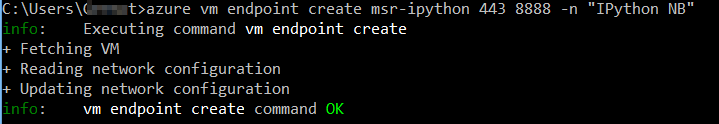


The IPython notebook runs on port 8888 in the default configuration. You can define any port as “public port”. A good choice might be port 443, the default port for HTTPS connections.

To create the endpoint, use the command:

azure vm endpoint create <vm-name> <public-port> <private-port> -n <name>

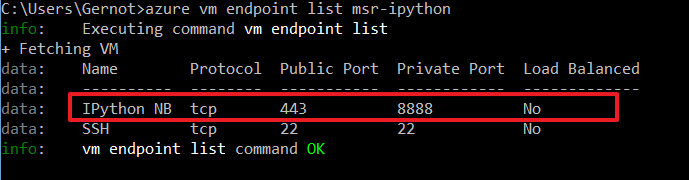
Use 443 for the public port, 8888 for the private port and an arbitrary name (for example “IPython NB”) as the name:



This takes a couple of moments. When you execute

azure vm endpoint list <vm-name>

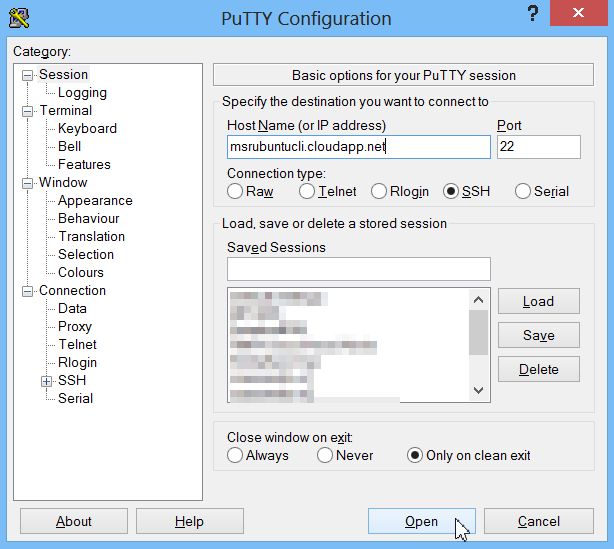
afterwards, you should be able to see the new entry:



# Initial System Maintenance

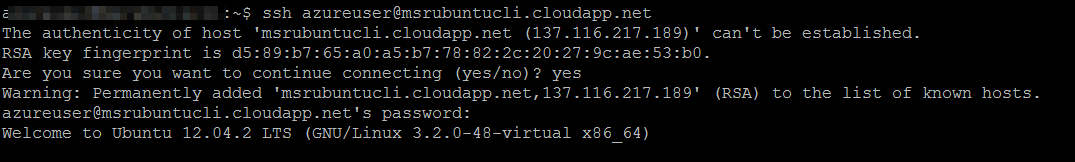
## Login to the VM

No matter if you created the VM from the portal or the CLI, you can login to it using SSH using host name <dns-name>.cloudapp.net and port 22. On Windows, you can use for example Putty to connect to it:



Of course, you can use a command line client as well:

ssh <username>@<dns-name>.cloudapp.net



## Upgrade Existing Packages

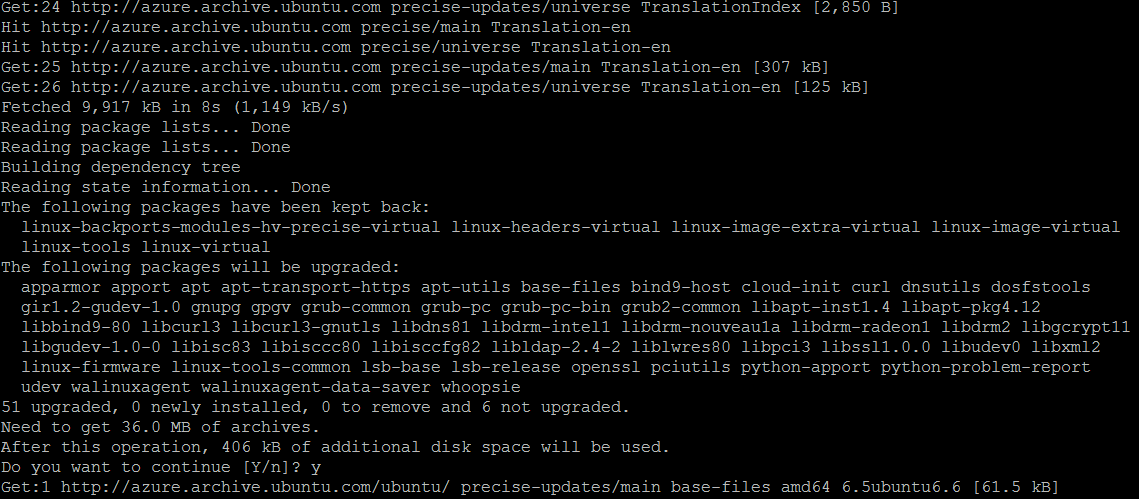
After you have connected to the virtual machine, you should update the system to incorporate the newest available software package versions. This should be done regularly as it both delivers security updates as well as bug fixes.

To update using the following command:

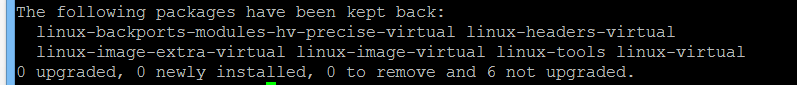
sudo apt-get update && sudo apt-get upgrade

The **sudo** command allows you to execute single commands as root (admin) user, prompting you for your password if you are authenticating via password. **apt-get update** will get an update of the available package list, while **apt-get upgrade** will upgrade all the software packages installed. The double ampersand **&&** means that the second command is only executed if the first command was successful.

The output should look this (excerpt):



Note: You will sometimes see the message that some packages have been kept back when performing the upgrade:



This happens when the dependencies of a package change. This often happens when there is a kernel upgrade available. You can upgrade these packages with the following command:

sudo apt-get dist-upgrade

This does not upgrade to a different release (as the name might imply), but incorporates a “smart” conflict resolution system for conflicting package dependencies. Use the command

man apt-get

to read about the dist-upgrade in more detail.

# Installing Python and Tools

## Installing Python

Python 2.7 is already installed on the Ubuntu image. There is a package called “python3” containing a python 3 installation, but this is not the standard distribution used in Ubuntu 12.04 LTS. This tutorial handles Python 2.7.

If you want to install Python 3, you can do so by typing

apt-get install python3

## Installing Easy Install and Pip

There are several ways you can get Python packages installed.  The easiest is probably by running the OS default installer, but sometimes it may not have the latest version in the version of Linux you are running.  For Ubuntu, **apt-get is the installer for the OS.  apt-get** will install your packages in /usr/lib/python/dist-packages.

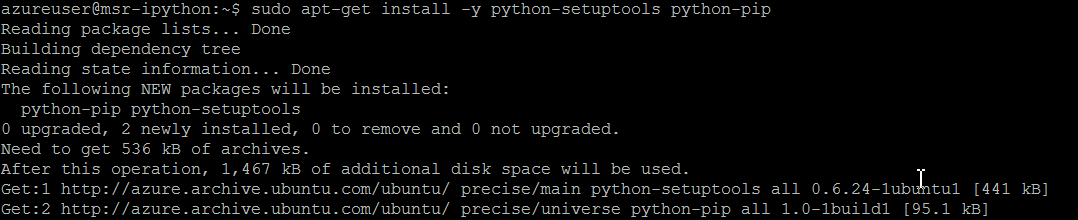
Another option is to use **easy\_install and/or pip.** They part of Python, not part of the Ubuntu OS.   We need to have python-setuptools package (for easy\_install) and python-pip (for pip) installed using apt-get first, before being able to use it.  If you use easy\_install or pip, all your packages will end up in /usr/local/lib/python/site-packages instead.

Usually, pip should be the recommended installation tool as it has better error messages, but you can simply use whichever tool you prefer (and even mix apt-get, easy\_install and pip).

You can install these two packages with the following command:

sudo apt-get install -y python-setuptools python-pip

This installs both packages. The -y flag avoids the “Do you want to continue” prompt during the installation process.

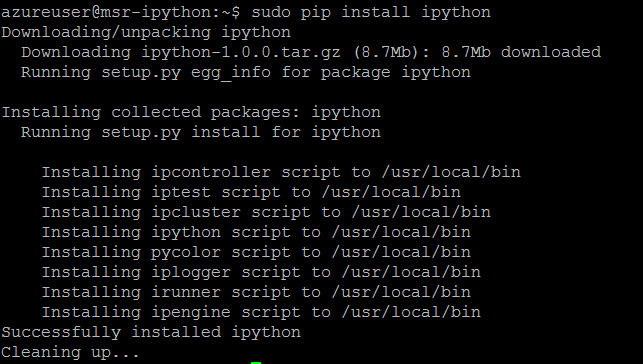


## Installing IPython

IPython provides a rich toolkit to help you make the most out of using Python interactively. Amongst other things, it includes an enhanced interactive shell, a web-based notebook environment as well as a high-performance library for parallel computing for multicore systems and cloud scenarios.

You can install it with pip using:

sudo pip install ipython



## Installing Additional Libraries

There are several useful additional libraries for python which you will install in this step. You can install them one by one or in one command (see the command at the end of this section).

Matplotlib is a popular 2D plotting library, it is one of IPython notebook’s component.  As you interact with the Notebook, plots are generated on the server using matplotlib and sent for displaying in your web browser. To install, type:

**sudo apt-get install -y python-matplotlib**

IPython notebook is browser based, it uses the **Tornado webserver**.  The Python-based Tornado webserver supports web sockets for interactive and efficient communication between the webserver and the browser.  To install, type:

sudo apt-get install -y python-tornado

We also need to install a package called Pyzmq, Zero MQ is a very fast networking package that IPython uses for its clustered configuration.  IPython is capable of interactively controlling a cluster of machines and run massively parallel  Big Compute and Big Data applications. To install it, type:

**sudo apt-get install -y python-zmq**

**Jinja2,** is a fast, modern and designer friendly templating language for Python is now required for IPython notebook. To install, type:

**sudo apt-get install –y python-jinja2**

**SciPy and NumPy contain numerical libraries that implement many features as for example Linear solvers. To install them, type:**

**sudo apt-get install -y python-scipy**

**Pandas is a data analysis libraries providing data structures (both labelled and unlabelled) that can be used with other scientific libraries as for example scipy and scikit-learn. To install it, type:**

**sudo apt-get install -y python-pandas**

**Scikit Learn is a fantastic python-based machine learning package, it includes algorithms for both supervised and unsupervised learning. Moreover, it includes support for sample datasets, data import tools, and model evaluation.**

**Scikit Learn is included with your Ubuntu distribution, but the default is about 2 versions behind. The best way to install Scikit Learn is to use PIP. This will build the packages from scratch and might take some time:**

**sudo pip install scikit-learn**

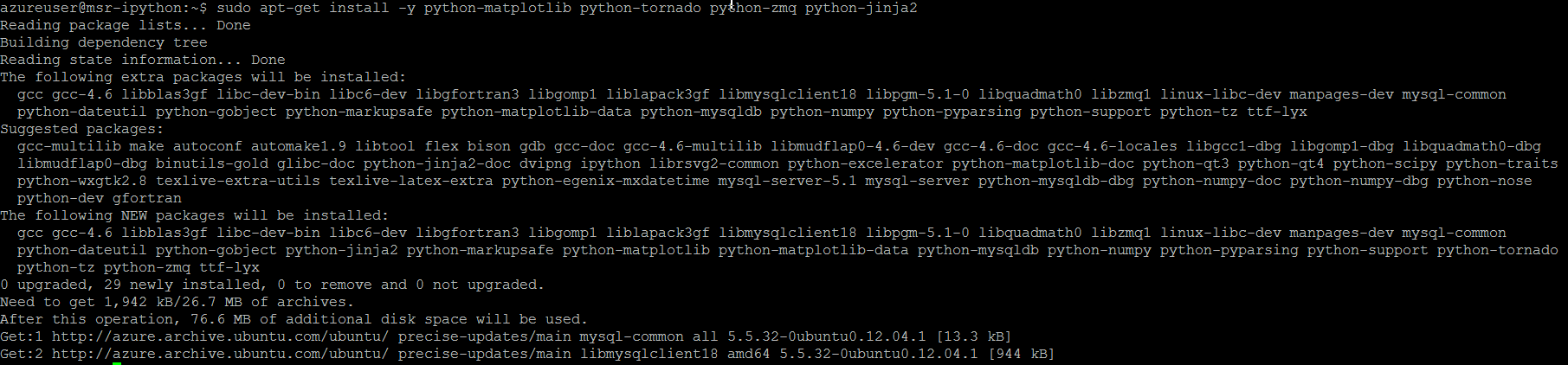
**And finally, to install the Azure SDK, type:**

**sudo pip install azure**

If you want to install all those libraries in one step, you can use the command:

sudo apt-get install -y python-matplotlib python-tornado python-zmq python-jinja2 **python-scipy python-pandas && sudo pip install azure scikit-learn**

This will install a number of dependencies and therefore can take some time:



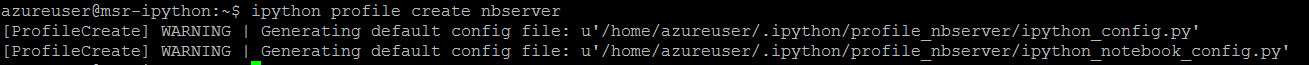
# Configuring IPython Notebook

## Creating Profile

Create a profile for the IPython notebook with the command:

ipython profile create nbserver

This creates a profile in your home directory under .ipython/profile\_nbserver/ipython\_config.py:



**Note that all directories and files starting with a dot are considered as hidden. To view them, you have to use**

**ls -al**

**On most Ubuntu installations, there is a preconfigured shortcut command named**

**ll**

(two lowercase Ls) that perform the same operation.

## Creating Credentials

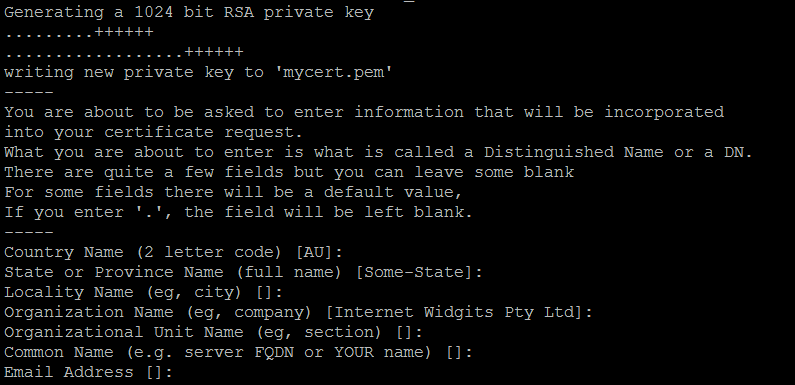
Next, create a certificate to allow an encrypted connection. First, change into the IPython profile directory you just created with:

cd ~/.ipython/profile\_nbserver/

Next, create a certificate using the command:

**openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mycert.pem -out mycert.pem**

You can leave the default information for the certificate or leave the default information:

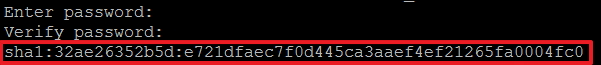


Since this is a self-signed certificate, the notebook your browser will give you a security warning. For long-term production use, you will want to use a properly signed certificate associated with your organization. Since certificate management is beyond the scope of this demo, we will stick to a self-signed certificate for now.

In the next step, you will create a hash of your desired password. You can do this with the following command:

**python -c "import IPython;print IPython.lib.passwd()"**

This operation will prompt you for your password and then print out the SHA1 hash of the password (storing plaintext passwords is discouraged). Remember the password’s hash:



## Configuring IPython Notebook

Now, you need to configure the IPython Notebook configuration. For this, you will start the editor nano to edit the configuration:

nano ipython\_notebook\_config.py

Copy the following content beneath the line “c = get\_config()”. Note that you should replace the password in this file with your password hash from above:

# This starts plotting support always with matplotlib

c**.**IPKernelApp**.**pylab **=** 'inline'

# You must give the path to the certificate file.

# If using a Linux VM:

c**.**NotebookApp**.**certfile **=** u'/home/azureuser/.ipython/profile\_nbserver/mycert.pem'

# Create your own password as indicated above

c**.**NotebookApp**.**password **=** u'sha1:32ae26352b5d:e721dfaec7f0d445ca3aaef4ef21265fa0004fc0' #use your own

# Network and browser details. We use a fixed port (9999) so it matches

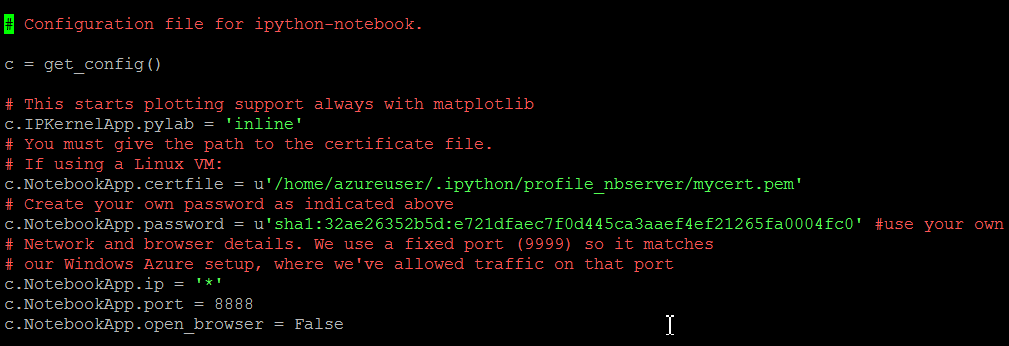
# our Windows Azure setup, where we've allowed traffic on that port

c**.**NotebookApp**.**ip **=** '\*'

c**.**NotebookApp**.**port **=** 8888

c**.**NotebookApp**.**open\_browser **=** **False**

With ctrl+X, you can save the file. You have to press “Y” to confirm the save operation and then press enter:

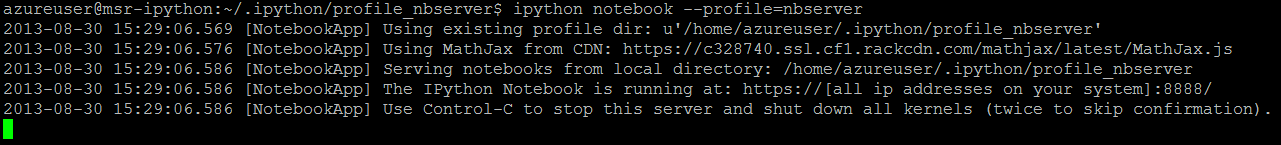


## Starting IPython Notebook

Now you can start IPython Notebook:

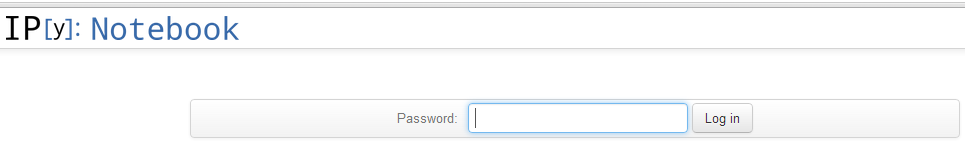
ipython notebook --profile=nbserver

You should see that the server has started:



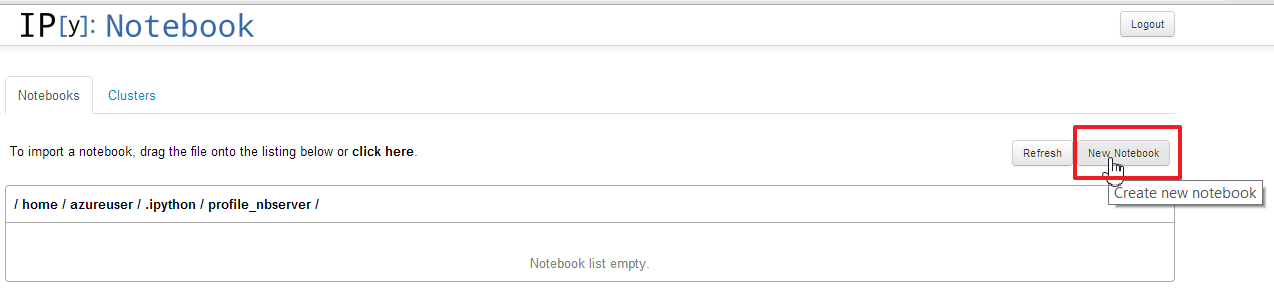
Navigate your browser to https://<vm-name>.cloudapp.net. Make sure that you use https and not http.

You will see a warning that the certificate is not signed. Since this is your own certificate, you can safely ignore this warning. After ignoring it, you should see the login screen:

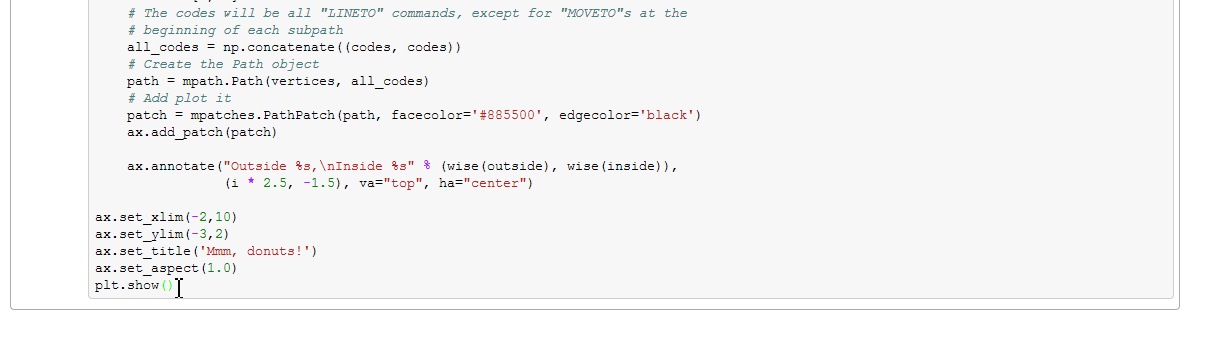


Login with the password you defined before (NOT the SHA1 hash).

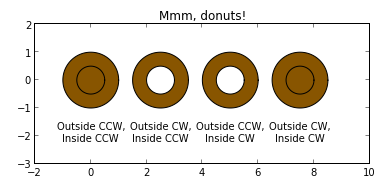
After you have logged in, click on “New Notebook”:



On the next screen, copy and paste the code from <http://matplotlib.org/examples/api/donut_demo.html> and place your cursor at the last line:



Then hit shift+enter. You should see the following result:



## Running IPython Notebook in Background

In the above example, your IPython Notebook while die once you close the SSH/Putty session. You can use a tool such as byobu or screen detach the server from your SSH session.

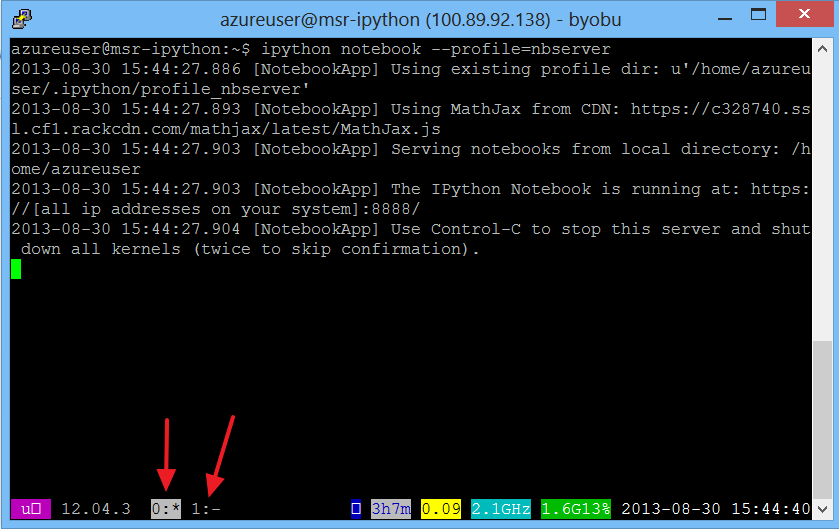
Stop the server process (ctrl + c) and enter the following command:

byobu

This will start a new shell that will stay alive even if you close your session. Start the server there with:

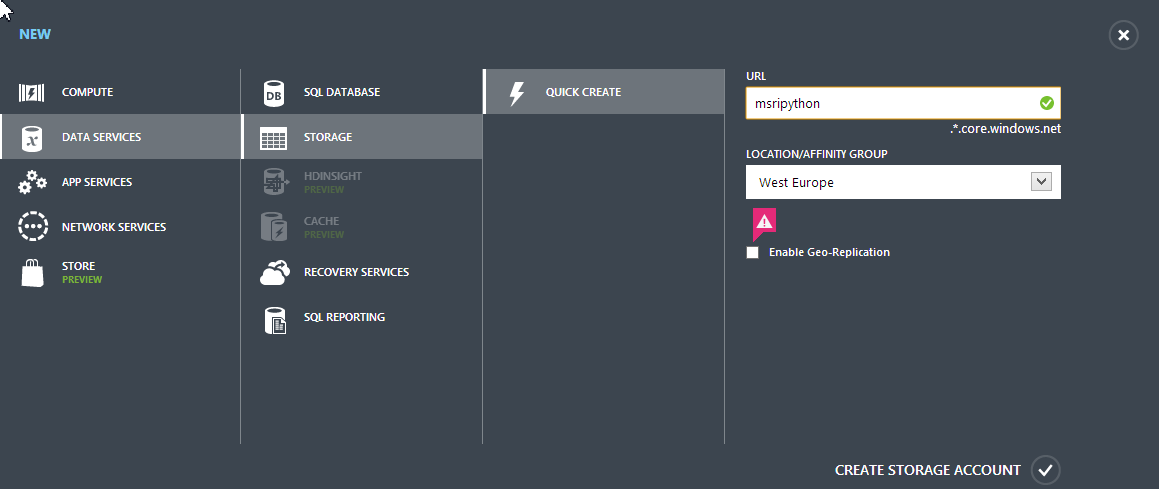
ipython notebook --profile=nbserver

Now, you can also open new shells with “F2” and switch between the shells with “F3” and “F4” on your keyboard. At the bottom, you will see the shell that is currently active. You can exit unnecessary shells with the exit command:

When pressing F6, you will detach from byobu, but it will stay active – even if you close your SSH session. Simply type “byobu” again to reattach to byobu.

# Access Azure Storage with Python

From the Azure management portal, select “New”/“Data Services”/”Storage”/”Quick Create”:

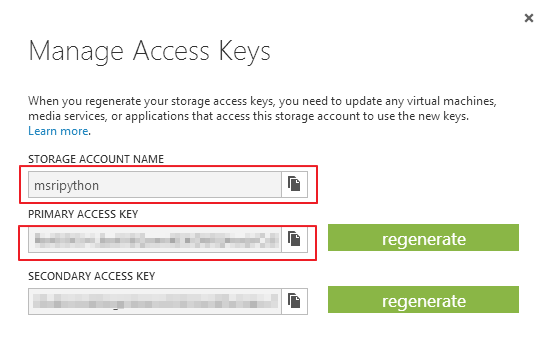


Choose a URL for the service and a location. Disable Geo Replicaiton, then click “Create Storage Account”.

Select your new storage account and choose “Manage access keys” at the bottom:



You will find your storage account name as well as access keys. Copy the account name and primary access key and save it for later use:



# Accessing Azure Storage from IPython Notebook

## Create Container

The first task is to create a container. You can do this with the following simple script:

**from** azure**.**storage **import** BlobService

account\_name **=** ""

account\_key **=** "" # REPLACE WITH YOUR KEY

blob\_service **=** BlobService**(**account\_name**,** account\_key**)**

blob\_service**.**create\_container**(**'data'**)**

The first line imports the BlobService class from the Azure Python SDK. You should replace lines 2 and 3 with your credentials from before.

Line 4 creates a blob service with your credentials. In line 5, you create a new container called “data”.

## Upload a File to Blob

You can upload a file into the new container with the following script:

**from** azure**.**storage **import** BlobService

account\_name **=** "" # REPLACE WITH YOUR ACCOUNT

account\_key **=** "" # REPLACE WITH YOUR KEY

file\_name **=** "clustering\_data.csv"

blob\_service **=** BlobService**(**account\_name**,** account\_key**)**

file\_content **=** file**(**file\_name**).**read**()**

blob\_service**.**put\_blob**(**'data'**,** 'clustering\_data'**,** file\_content**,** 'BlockBlob'**)**

Again, define your account name and account key. In line 4, define the name of the file you want to upload. File 6 reads in the file and file 7 puts it into the container “data” as “clustering\_data”

## Read Contents from Blob

To read contents of a file in blob storage, use the following script:

**from** azure**.**storage **import** BlobService

account\_name **=** "msripython" # REPLACE WITH YOUR ACCOUNT

account\_key **=** "ReXK90J+L6zxlX6QswoKEKGkRQ4ovjVOJZM71lw8DGfmf9+ExVlt92IgSJ3AH451RnjrKgfNbtDhHS2cd7iSLQ==" # REPLACE WITH YOUR KEY

blob\_service **=** BlobService**(**account\_name**,** account\_key**)**

content **=** blob\_service**.**get\_blob**(**'data'**,** 'clustering\_data'**)**

**print** content

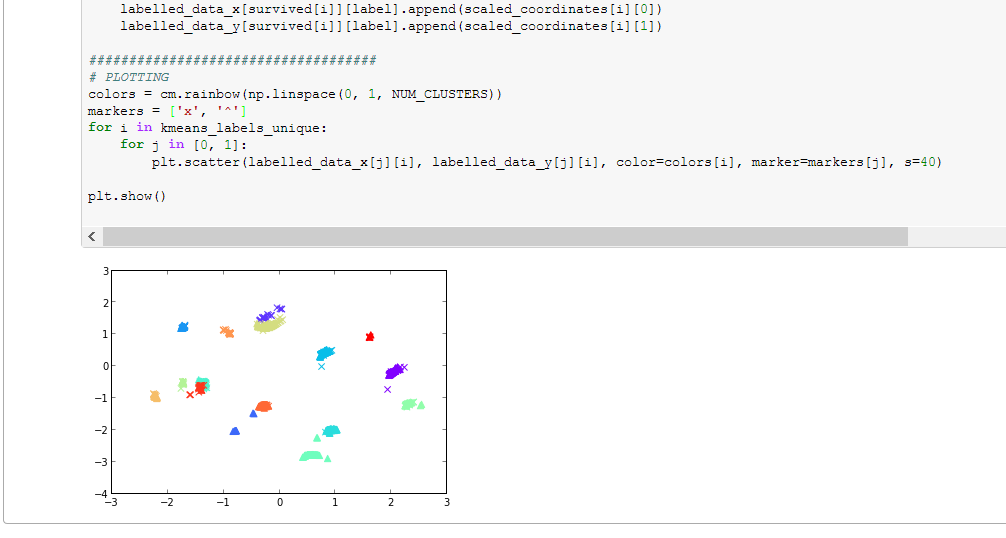
This gets the blob data and prnts the content (last two lines).

# Clustering Example with Pandas and Scikit-learn

The following example (cluster-titanic.py) clusters passengers of the titanic based on several attributes. It is taken from [www.kaggle.com](http://www.kaggle.com) and has been preprocessed with Excel and then exported in CSV format.

The example assumes that the file “clustering\_data” in the container “data” contains the test data from titanic-data.csv.

The script will result in an output as:



In the beginning, the script defines the azure credentials as well as the intended number of clusters. The data is loaded from azure storage and stored in a tempfile:

####################################

# Get data from azure blob and write it to temp file

blob\_service **=** BlobService**(**account\_name**,** account\_key**)**

content **=** blob\_service**.**get\_blob**(**'data'**,** 'clustering\_data'**)**

**with** open**(**"tmpfile"**,** "w"**)** **as** f**:**

f**.**write**(**content**)**

In the next step, the data set is loaded with pandas. Pandas is a library that makes working with data tables as for example CSV data easy. As the “names” and “survived” groups are not needed for the clustering, they are removed from the data frame:

####################################

# DATA LOADING AND PREPARATION

# Load data as pandas dataframe

data **=** pandas**.**io**.**parsers**.**read\_csv**(**'tmpfile'**,** sep**=**";"**)**

# Remove name and survived dimension to learn

names **=** data**.**pop**(**'name'**)**

survived **=** data**.**pop**(**'survived'**)**

Afterwards, the KMeans clustering operation is initialized, the algorithm is trained and the results (labels for each data set, cluster centers and the set of labels used) are stored in the appropriate variables:

####################################

# CLUSTERING

# Create KMeans

kmeans **=** KMeans**(**n\_clusters**=**NUM\_CLUSTERS**,** init**=**'k-means++'**,** n\_init**=**10**,** max\_iter**=**300**,** tol**=**0.0001**,** precompute\_distances**=True,** verbose**=**0**,** random\_state**=None,** copy\_x**=True,** n\_jobs**=**1**)**

# Train KMeans

kmeans**.**fit**(**data**)**

# Get the results

kmeans\_labels **=** kmeans**.**labels\_

kmeans\_cluster\_centers **=** kmeans**.**cluster\_centers\_

kmeans\_labels\_unique **=** np**.**unique**(**kmeans\_labels**)**

After clustering the data, the multi-dimensional data is reduced to two dimensions in order to allow plotting:

####################################

# PLOT PREPARATION

# Reduce to two dimensions for plotting

mds **=** MDS**(**n\_components**=**2**)**

mds**.**fit**(**data**)**

scaled\_coordinates **=** mds**.**embedding\_

The script prepares the data in order to be plotted in multiple colors (depending on their cluster) as well as the status of whether the passenger has survived or not. Survived passengers and those who did not survive are assigned different markers and in the end, the plot is shown:

####################################

# PLOTTING

colors **=** cm**.**rainbow**(**np**.**linspace**(**0**,** 1**,** NUM\_CLUSTERS**))**

markers **=** **[**'x'**,** '^'**]**

**for** i **in** kmeans\_labels\_unique**:**

**for** j **in** **[**0**,** 1**]:**

plt**.**scatter**(**labelled\_data\_x**[**j**][**i**],** labelled\_data\_y**[**j**][**i**],** color**=**colors**[**i**],** marker**=**markers**[**j**],** s**=**40**)**

plt**.**show**()**

