**Question1:**

We have below JSON and it has name column in the root as well as in the delivery struct. We have to rename the name element to avoid ambiguous.

**Input1.json:**

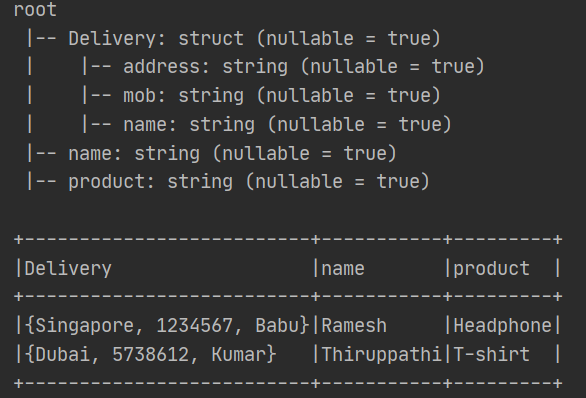
[{"name":"Ramesh",  
 "product":"Headphone",  
 "Delivery":{  
 "name":"Babu",  
 "address":"Singapore",  
 "mob":"1234567"}},  
{"name":"Thiruppathi",  
 "product":"T-shirt",  
 "Delivery":{  
 "name":"Kumar",  
 "address":"Dubai",  
 "mob":"5738612"}}  
]

**Solution:**

**Flatten the JSON sub element:** When we have the sub element like delivery in the json we can choose that using delivery.\* to get it flattened as shown below.

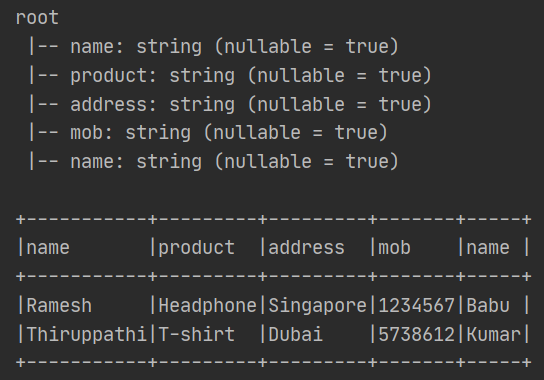
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.master("local[1]").appName("Ambiguous").getOrCreate()  
df = spark.read.format("json")\  
 .option("multiLine", "True")\  
 .load("../input/input1.json")  
df.printSchema()  
df.show(truncate=False)

Output:



**Flatten the Delivery Column:**

df = df.select("\*", "Delivery.\*").drop("Delivery")  
df.printSchema()  
df.show(truncate=False)

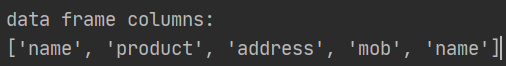


Now we have two name column in the schema

**Solution:**

1. Identify the index of the duplicate column in the data frame and store that value in the list. In our name index 0 and 4 (name column)
2. Then iterate the above list and change the column name in the data frame after appending the index value

df\_cols = df.columns # this will return the list  
print("data frame columns:")  
print(df\_cols)

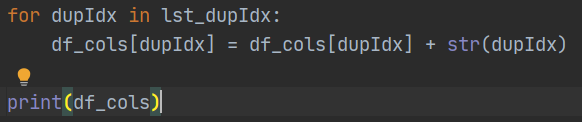


Duplicate index:

lst\_dupIdx = []  
idx = 0  
for col\_name in df\_cols:  
 if df\_cols.count(col\_name) >= 2:  
 lst\_dupIdx.append(idx)  
 idx = idx + 1  
  
print("dup index")  
print(lst\_dupIdx)



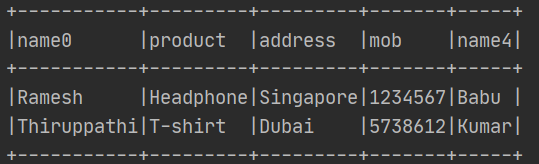
Rename the columns in the data frame



]

Now use that to select from data frame

# Note the toDF accept the string https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrame.toDF.html  
df = df.toDF(\*df\_cols)   
df.show(truncate=False)



**Complete Code:**

#ref https://www.youtube.com/watch?v=SGRM3u\_Fk8s&list=PLY6Ag0EOw54yWvp\_hmSzqrKDLhjdDczNC&index=21  
  
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.master("local[1]").appName("Ambiguous").getOrCreate()  
df = spark.read.format("json")\  
 .option("multiLine", "True")\  
 .load("../input/input1.json")  
df.printSchema()  
df.show(truncate=False)  
df = df.select("\*", "Delivery.\*").drop("Delivery")  
df.printSchema()  
df.show(truncate=False)  
#df.select("name").show() #this will throw error as name column is available twice  
#steps to remove the ambiguity columns  
#1.store the column name in the list  
#2.iterate the column name and check for which column name present more than once and add that index into the list  
#3.iternate the list created in the prev step and change the column name in the column name list by adding the index  
#4.now use that column name to rename the df name  
  
df\_cols = df.columns # this will return the list  
print("data frame columns:")  
print(df\_cols)  
lst\_dupIdx = []  
idx = 0  
for col\_name in df\_cols:  
 if df\_cols.count(col\_name) >= 2:  
 lst\_dupIdx.append(idx)  
 idx = idx + 1  
  
print("dup index")  
print(lst\_dupIdx)  
  
for dupIdx in lst\_dupIdx:  
 df\_cols[dupIdx] = df\_cols[dupIdx] + str(dupIdx)  
  
print(df\_cols)  
  
# Note the toDF accept the string https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrame.toDF.html  
df = df.toDF(\*df\_cols)  
df.show(truncate=False)

()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()

**Question2:** what is toDF method in pyspark

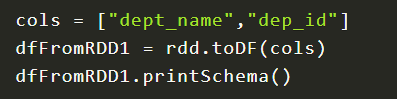
The pyspark.sql.DataFrame.toDF() function is used to create the data frame with the specified column name. It creates the DataFrame from RDD. Since the RDD is schema less without column name and data type, converting RDD to a DataFrame gives you default column names as \_1, \_2 and soon and a data type as string.

**Key points about toDF:**

1. toDF() returns a DataFrame
2. The toDF() is present on both RDD and DataFrame data structure
3. The toDF() by default create column name as \_1, \_2 and soon
4. toDF() also support taking column name as a list or schema as a argument

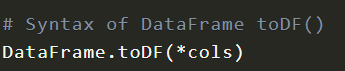
**Using toDF with RDD:**

When we are using toDF() with RDD we have to pass the column name as list to toDF method



**Using toDF with DataFrame:**

When we are using toDF() with DataFrame we have pass the name as \*list to toDF method



The function is used to set the column names when your dataframe contains the default name or change the column name of the entire dataframe.

**What happens when we apply \* infront of list:**

It returns the element as string separated by space.

**Using \* operator to unpack function:**

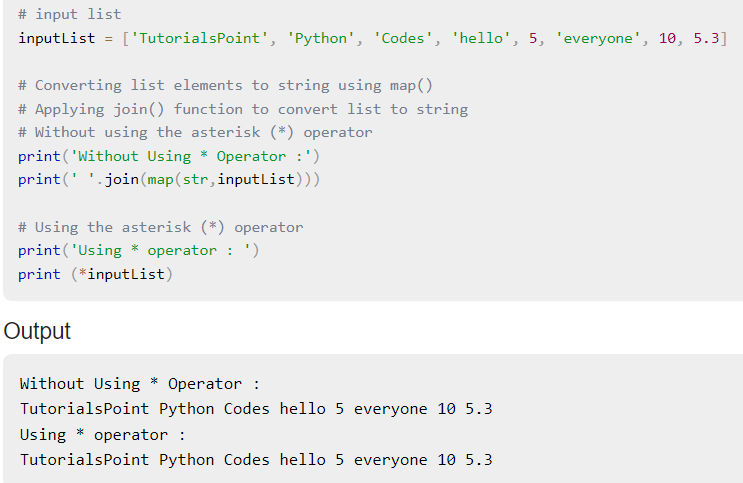
Ref - <https://www.tutorialspoint.com/How-does-operator-work-on-list-in-Python#:~:text=Python%20List%20also%20includes%20the,the%20specified%20number%20of%20times>.

This method is quite handy when printing data in a raw format (without any commas and brackets ). Many programmers attempt to remove commas and brackets by converging functions, thus this simple prefix asterisk can fix your problem in unpacking them.

Algorithm Steps:

Following are the Algorithm/steps to be followed to perform the desired task −

* Create a variable to store the input list and give it some random values.
* To print list elements separated by spaces without brackets [] first we convert the list to a string by passing the str and list as arguments to the map () function. It converts each element of the list to the string type and returns the resultant list of items. The join() function(join() is used to join elements of a sequence that are separated by a string separator) is used to convert the result list to a string.
* Instead of the previous method, we can use the asterisk operator (\*) to print the list separated by spaces



()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()

**Pyspark remove special characters in column name:**

**Input File:**

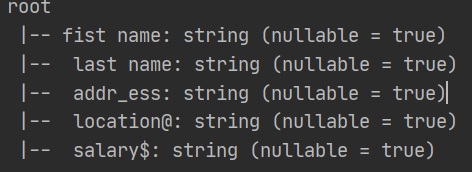
fist name, last name, addr\_ess, location@, salary$  
thiruppathi, s, singapore, singapore, 22000

from pyspark.sql import SparkSession  
import pyspark.sql.functions as F  
import re

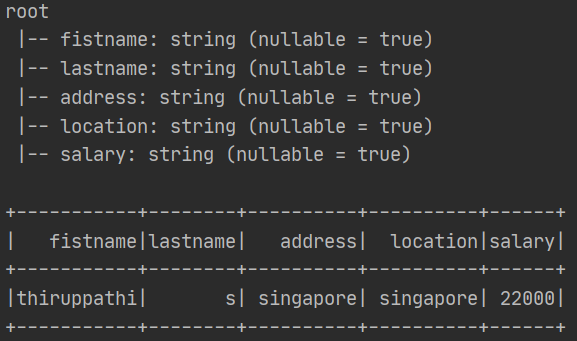
**Approach1:**

We can use the alias function while select to rename the column as shown below. While using alias iterate the dataframe column and remove the special characters.

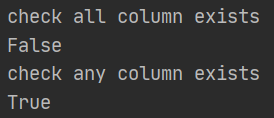
spark = SparkSession.builder.master("local[1]").appName("check\_col\_exists").getOrCreate()  
df = spark.read.format("csv").option("header",True).option("delimiter",",").load("../input/employee.txt")  
print(df.printSchema())



df=df.select([F.col(colname).alias(re.sub("[^a**-**z,A**-**Z,0**-**9]+","",colname)) for colname in df.columns])  
df.printSchema()  
df.show()



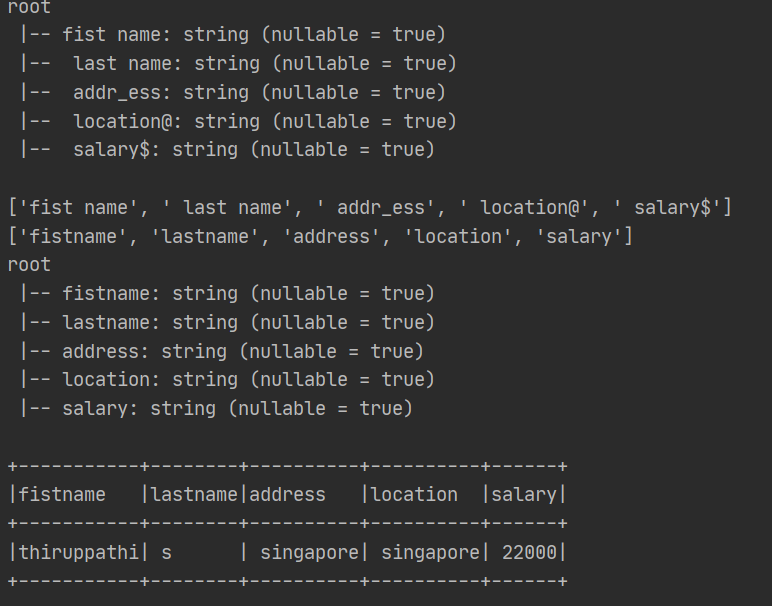
validate\_cols=["location","esalary"]  
print("check all column exists")  
if all([c in df.columns for c in validate\_cols]):  
 print("True")  
else:  
 print("False")  
  
print("check any column exists")  
if any([c in df.columns for c in validate\_cols]):  
 print("True")  
else:  
 print("False")



**Approach2:**

**Using toDF method**

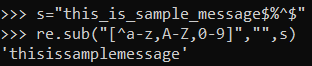
1. Get the column name as list
2. Do the required transformation in the list
3. Use dataFrame.toDF() method to rename the column
4. spark = SparkSession.builder.master("local[1]").appName("check\_col\_exists").getOrCreate()  
   df = spark.read.format("csv").option("header",True).option("delimiter",",").load("../input/employee.txt")  
   df.printSchema()  
     
   #this returns list  
   col\_name=df.columns  
   print(col\_name)  
   col\_name = list(map(lambda c : c.replace("@","").replace("$","").replace("\_","").replace(" ",""), col\_name))  
   print(col\_name)  
   df2=df.toDF(\*col\_name)  
   df2.printSchema()  
   df2.show(truncate=False)



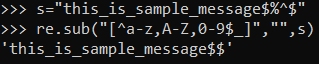
**RegExp Function:**

Assume we need to keep only the alphanumeric values when we have to do the following. We have to use re.sub function the last parameter is the source and second parameter is the value to replace when other value found. The first parameter is the value that needs to be keep, if we provide range (like a-z and A-Z) then we have to separate those with comma and inside square bracket we have to start with ^.

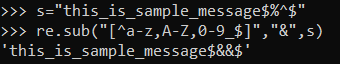
Example – Keep only alphanumeric remove all the sepecial character



Example – Keep only alphanumeric and underscore and dollar symbol



Example – Keep only alphanumeric and underscore and dollar symbol and replace other not allowed characters with &



()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()

**Spark Submit Command:**

The –master decides where this application will be running

* Local – this application will be running in local, by default it will run in single core
* Local[2] – This application will be running in local with 2 cores
* Local[\*] – This application will be running in local and use all the available 2 cores
* Yarn – to run the application in YARN cluster

The deploy mode – specifies whether the application would be running in client mode or cluster mode. This accepts two values client and cluster, the default value is client.

()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()

**Manage Python Dependencies in Cluster:**

<https://spark.apache.org/docs/latest/api/python/user_guide/python_packaging.html>

There are multiple ways to manage python dependencies in cluster:

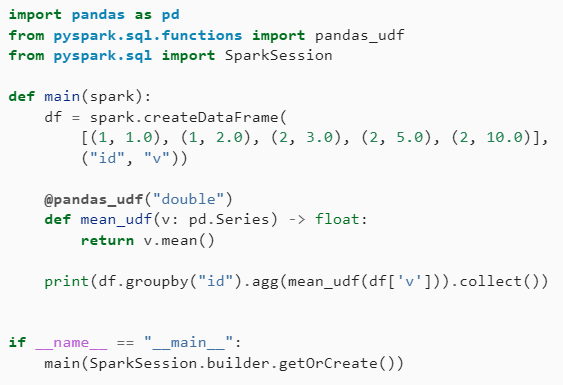
1. Using pyspark native features
2. Using conda
3. Using Virtualenv
4. Using PEX

**Problem Statement:**

When you want to run your PySpark application on a cluster such as YARN, Kubernetes, Mesos, etc., you need to make sure that your code and all used libraries are available on the executors.

As an example let’s say you may want to run the Pandas UDF’s examples. As it uses pyarrow as an underlying implementation we need to make sure to have pyarrow installed on each executor on the cluster. Otherwise you may get errors such as ModuleNotFoundError: No module named 'pyarrow'.

Here is the script app.py from the previous example that will be executed on the cluster:



**Using Pyspark Native Features:**

PySpark allows to upload Python files (.py), zipped Python packages (.zip), and Egg files (.egg) to the executors by one of the following:

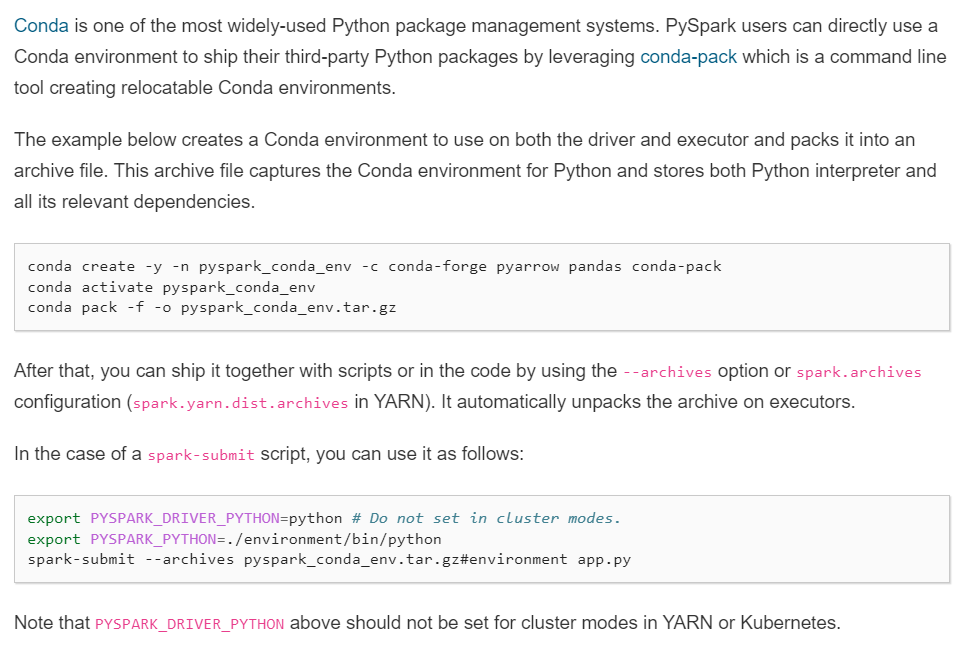
* Setting the configuration setting spark.submit.pyFiles
* Setting --py-files option in Spark scripts
* Directly calling pyspark.SparkContext.addPyFile() in applications

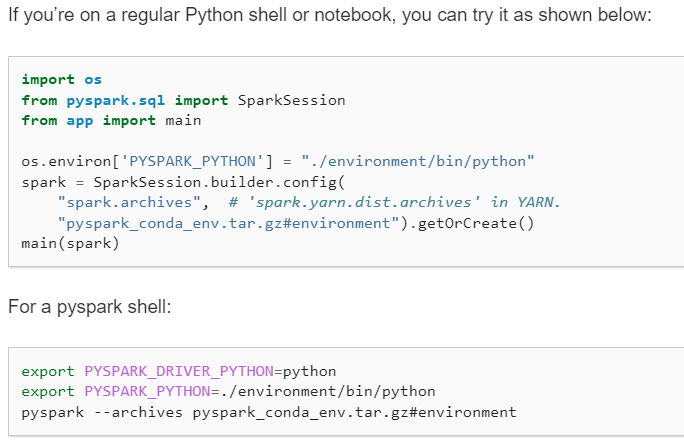
This is a straightforward method to ship additional custom Python code to the cluster. You can just add individual files or zip whole packages and upload them. Using pyspark.SparkContext.addPyFile() allows to upload code even after having started your job.

However, it does not allow to add packages built as Wheels and therefore does not allow to include dependencies with native code.

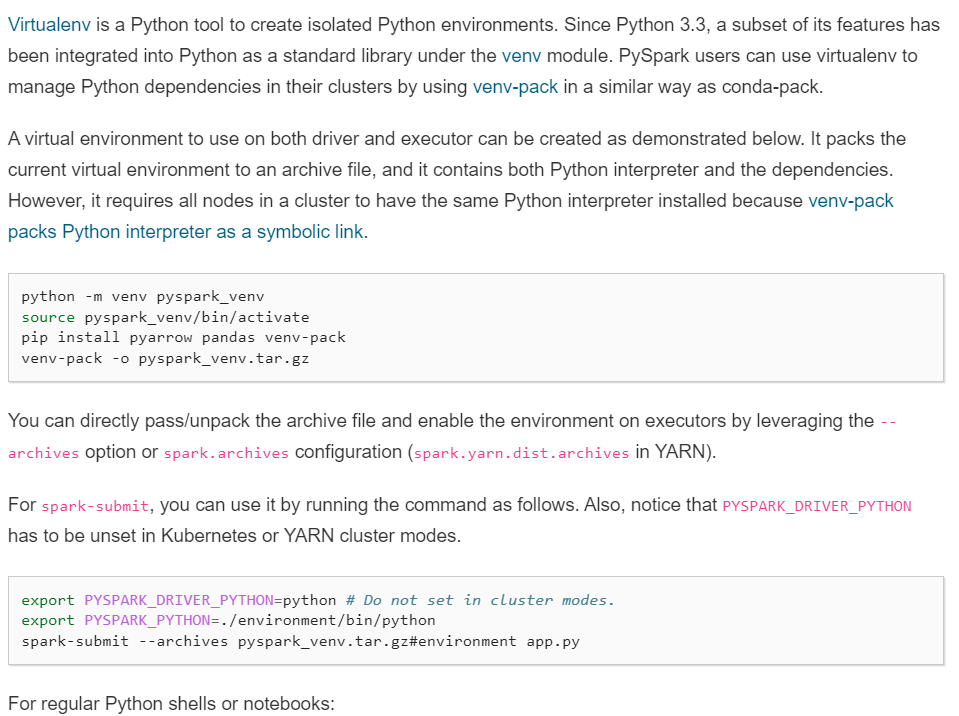
<https://www.databricks.com/blog/2020/12/22/how-to-manage-python-dependencies-in-pyspark.html>

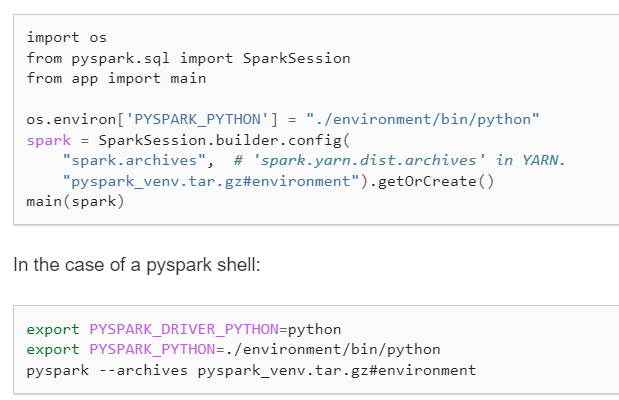
**Using Conda:**



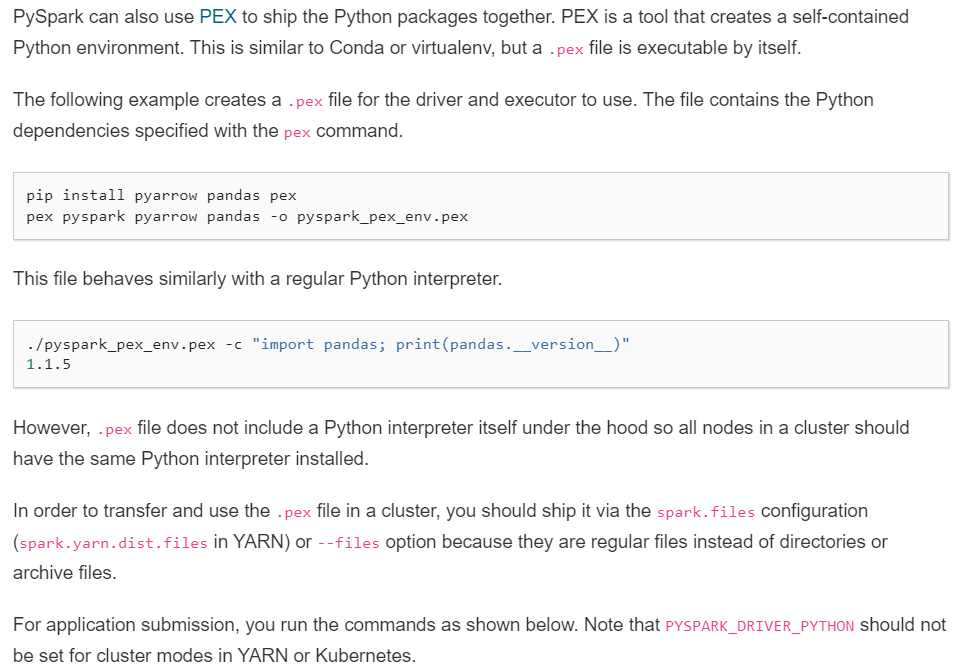


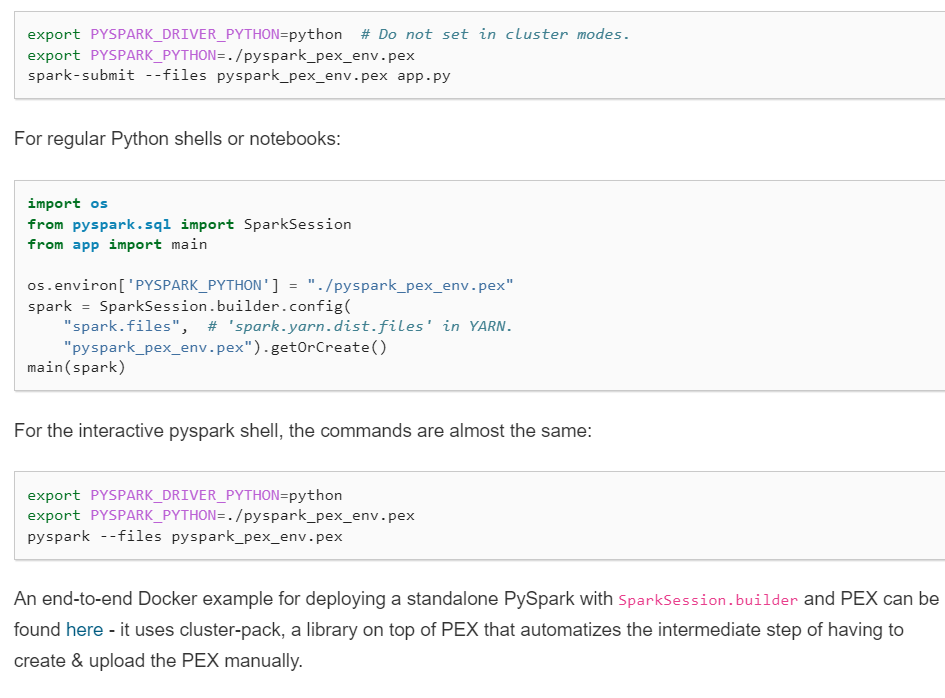
**Using Virtualenv:**





Using PEX:





Note: These package management systems can handle any Python packages that --py-files or spark.submit.pyFiles configuration cannot cover. Users can seamlessly ship not only pandas and PyArrow but also other dependencies to interact together when they work with PySpark.

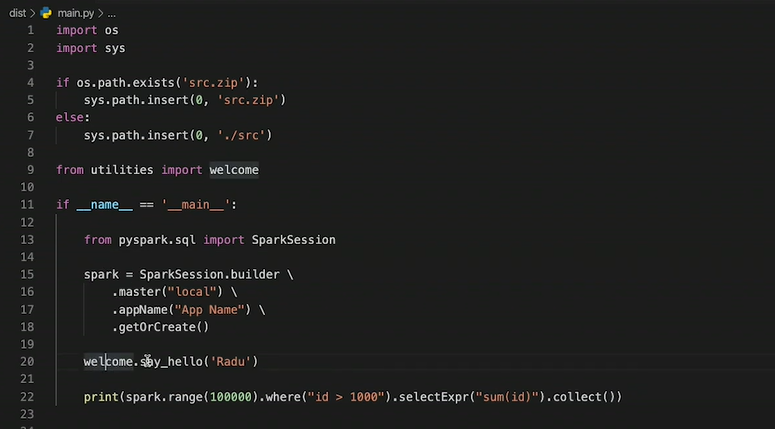
Managing Dependencies and Artifacts in PySpark:

<https://bytes.grubhub.com/managing-dependencies-and-artifacts-in-pyspark-7641aa89ddb7>

()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()

**How to structure your pyspark jobs and code:**

<https://www.youtube.com/watch?v=RAT-gQccsEs&t=147s>



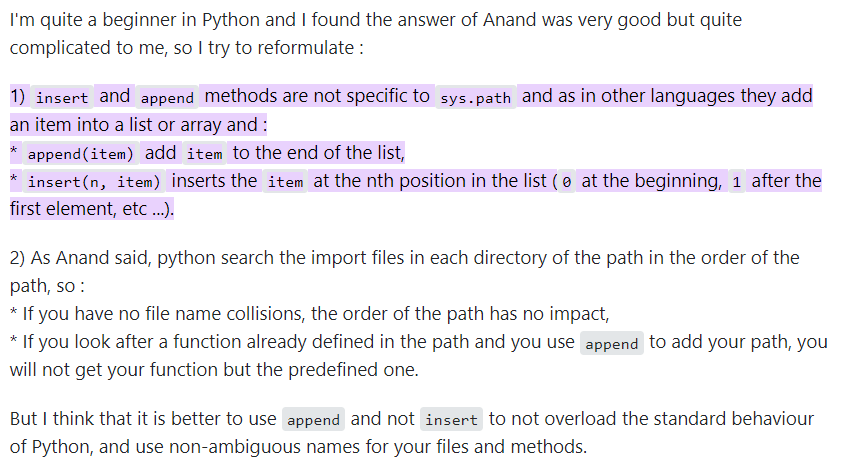
sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search in for the required module.

When a module(a module is a python file) is imported within a Python file, the interpreter first searches for the specified module among its built-in modules. If not found it looks through the list of directories(a directory is a folder that contains related modules) defined by sys.path.

**Sys.path.insert vs Sys.path.append:**

I was recently having a problem with a python ImportError, where the module was found when running on my local computer but not found on the CI server. I solved this problem by swapping sys.path.append(path) in my script with sys.path.insert(0, path) where path is the string module location.

Since this is my module and not an installed package (related question), why does the order of paths fix this problem?



()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()()