# INITIATING SPARK SESSION & DATASET CONNECTIONS  
   
from marvel.framework.files.read import cnt  
 from pyspark.sql import HiveContext, SparkSession  
 from pyspark.sql import functions as F  
   
spark = SparkSession \  
 .builder \  
 .appName("APP\_NAME\_MarvelEXT\_as") \  
 .enableHiveSupport() \  
 .getOrCreate()  
   
# import static references for hdfs from 'py4j.java\_gateway.JavaClass'> (conf, filesystem, path, fileutil)  
   
sc = spark.sparkContext  
   
fileutil = sc.\_jvm.org.apache.hadoop.fs.FileUtil  
 hdfs\_path = sc.\_jvm.org.apache.hadoop.fs.Path  
   
filesystem = sc.\_jvm.org.apache.hadoop.fs.FileSystem  
 jvm\_conf = sc.\_jsc.hadoopConfiguration()  
 hdfs\_fs = filesystem.get(jvm\_conf)  
   
# set jvm\_sys\_properties to > overwrite - dynamic  
   
spark.conf.set("hive.exec.dynamic.partition", "true")  
 spark.conf.set("hive.exec.dynamic.partition.mode", "nonstrict")  
 spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")  
   
  
# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*#  
   
class dtutil:  
   
 # retaining JVM spark native date formats, and not converting from python date formats  
   
 def \_\_init\_\_(self):  
 dat\_du = [[1]]  
 col\_du = ["row\_id"]  
 self.df\_dtutil = spark.createDataFrame(dat\_du, col\_du)  
   
 def todaystr(self):  
 df\_du\_today = self.df\_dtutil  
 df\_du\_today = df\_du\_today.withColumn("curr\_date", F.current\_date())  
 curr\_date\_str = str(df\_du\_today.first().curr\_date)  
   
 return curr\_date\_str  
   
  
# end of class dtutil  
   
  
# ~~~ START <jobnm #1 :x\_writer > ~~~ #  
   
class x\_writer:  
   
 def \_\_init\_\_(self, report\_nm):  
   
 self.rptnm = report\_nm  
   
 # INITIALISE SOURCES & DESTINATION  
   
 self.target\_DB\_ext = cnt['Customer\_Staging\_Mart']['cdd\_ext\_database']['ext\_marv\_database'] #"gb\_customer\_data\_domain\_secured\_rpt"  
   
 # rptnm\_list = ["la\_acct", "cr\_earn", "cr\_burn"]  
   
 if self.rptnm == "la\_acct":  
 self.source\_DB\_rpt = cnt['Customer\_Staging\_Mart']['cdd\_rpt\_database']['secured\_reporting\_database'] #"gb\_customer\_data\_domain\_secured\_rpt"  
 self.source\_TB = cnt['Customer\_Staging\_Mart']['csm\_rpt\_tables']['rpt\_loyalty\_account'] #"cdd\_rpt\_loyalty\_acct"  
 self.target\_TB\_ext = cnt['Customer\_Staging\_Mart']['csm\_ext\_tables']['ext\_as\_loyalty\_account'] #"as\_la\_acct\_ext\_tb"  
   
 if self.rptnm == "cr\_earn":  
 self.source\_DB\_rpt = cnt['Customer\_Staging\_Mart']['cdd\_rpt\_database']['rpt\_database'] #"gb\_customer\_data\_domain\_rpt"  
 self.source\_TB = cnt['Customer\_Staging\_Mart']['csm\_rpt\_tables']['rpt\_campaign\_reward\_earn'] #"cdd\_rpt\_cmpgn\_reward\_earn"  
 self.target\_TB\_ext = cnt['Customer\_Staging\_Mart']['csm\_ext\_tables']['ext\_as\_campaign\_reward\_earn'] #"as\_cr\_earn\_ext\_tb"  
   
 if self.rptnm == "cr\_burn":  
 self.source\_DB\_rpt = cnt['Customer\_Staging\_Mart']['cdd\_rpt\_database']['rpt\_database'] #"gb\_customer\_data\_domain\_rpt"  
 self.source\_TB = cnt['Customer\_Staging\_Mart']['csm\_rpt\_tables']['rpt\_campaign\_reward\_burn'] #"cdd\_rpt\_cmpgn\_reward\_burn"  
 self.target\_TB\_ext = cnt['Customer\_Staging\_Mart']['csm\_ext\_tables']['ext\_as\_campaign\_reward\_burn'] #"as\_cr\_burn\_ext\_tb"  
   
  
  
 def db\_write(self):  
   
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* SPARK JOB Inititated for <jobnm:DATATO\_target.dbnm.tbnm>\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
   
 # READ DATASETS  
   
 if self.rptnm == "la\_acct":  
 # process report specific conditions  
 column\_str\_la\_acct = "\*"  
 query\_as = "select " + column\_str\_la\_acct + " from {}.{} where acct\_status\_id = 'Full'"  
 df\_ext\_as = spark.sql(query\_as.format(self.source\_DB\_rpt, self.source\_TB))  
 df\_ext\_as = df\_ext\_as.filter(df\_ext\_as.push\_notif\_consent == "Y")  
 print("Initiating la\_acct db writing")  
   
 if self.rptnm == "cr\_earn":  
 # process report specific conditions  
 column\_str\_cr\_earn = "\*"  
 query\_as = "select " + column\_str\_cr\_earn + " from {}.{} where acct\_status\_id = 'Full'"  
 df\_ext\_as = spark.sql(query\_as.format(self.source\_DB\_rpt, self.source\_TB))  
 print("Initiating cr\_earn db writing")  
   
 if self.rptnm == "cr\_burn":  
 # process report specific conditions  
 column\_str\_cr\_burn = "\*"  
 query\_as = "select " + column\_str\_cr\_burn + " from {}.{} where acct\_status\_id = 'Full'"  
 df\_ext\_as = spark.sql(query\_as.format(self.source\_DB\_rpt, self.source\_TB))  
 print("Initiating cr\_burn db writing")  
   
 df\_ext\_as = df\_ext\_as.withColumn("extract\_date", F.current\_date())  
 # df\_ext\_as.show(50)  
   
 # WRITE DATASETS  
 # write dataframe into external table with "overwrite partition" mode, set to dynamic  
 # MetaStore Meta Entries > database = "marvel\_ext\_db" , TB = target\_TB\_ext\_la, Location = path\_to\_dbtb  
   
 hdfs\_locale = "/user/svc\_uk\_cust\_rdl/marvel\_extracts"  
 path\_to\_dbtb = hdfs\_locale + "/" + self.target\_DB\_ext + "/" + self.target\_TB\_ext  
   
 print(path\_to\_dbtb)  
   
 partition\_key = "extract\_date"  
   
 if (spark.\_jsparkSession.catalog().tableExists(self.target\_DB\_ext + "." + self.target\_TB\_ext)):  
 df\_ext\_as.write.mode("overwrite").insertInto(self.target\_DB\_ext + "." + self.target\_TB\_ext, overwrite=True)  
 print("Table " + self.target\_DB\_ext + "." + self.target\_TB\_ext + " exits... OVERWRITE Insert Into New records in Partition for Today")  
 else:  
 print("creating Table " + self.target\_DB\_ext + "." + self.target\_TB\_ext + " for the very FIRST time")  
 df\_ext\_as.write \  
 .mode("overwrite") \  
 .partitionBy(partition\_key) \  
 .option("path", path\_to\_dbtb) \  
 .saveAsTable("{}.{}".format(self.target\_DB\_ext, self.target\_TB\_ext))  
 # endif  
   
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* SPARK JOB complete for <jobnm>\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
   
  
# end of class  
   
# ~~~ FINISH <jobnm #1:x\_writer> ~~~ #  
   
  
# ~~~ START <jobnm #2 :x\_slicer > ~~~ #  
   
class x\_slicer:  
   
 def \_\_init\_\_(self, report\_nm):  
   
 self.rptnm = report\_nm  
 # INPUTS  
   
 self.source\_DB\_ext = "marvel\_ext\_db"  
   
 # rptnm\_list = ["la\_acct", "cr\_earn", "cr\_burn"]  
   
 if self.rptnm == "la\_acct":  
 self.source\_TB\_ext = "as\_la\_acct\_ext\_tb"  
 self.column\_str = """singl\_profl\_id, first\_nm, chnl\_type, date\_format(regexp\_replace(wallet\_create\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as wallet\_create\_ts, curr\_cash\_pot\_pnts\_qty, voucher\_live\_amt, voucher\_live\_cnt, voucher\_spent\_amt, voucher\_spent\_cnt, date\_format(regexp\_replace(last\_loyalty\_scan\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as last\_loyalty\_scan\_ts, total\_nbr\_of\_scans, last\_shopped\_store\_id, date\_format(regexp\_replace(last\_reward\_created\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as last\_reward\_created\_ts, total\_cmpgns\_cmplt, date\_format(regexp\_replace(last\_reward\_used\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as last\_reward\_used\_ts, date\_format(regexp\_replace(last\_mssn\_prgrss\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as last\_mssn\_prgrss\_ts, date\_format(regexp\_replace(last\_mssn\_cmplt\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as last\_mssn\_cmplt\_ts, total\_mssns\_started, total\_mssns\_live"""  
   
 if self.rptnm == "cr\_earn":  
 self.source\_TB\_ext = "as\_cr\_earn\_ext\_tb"  
 self.column\_str = """singl\_profl\_id, loyalty\_id, earn\_cmpgn\_id, mssn\_id, date\_format(regexp\_replace(cmpgn\_seen\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as cmpgn\_seen\_ts, date\_format(regexp\_replace(cmpgn\_cmplt\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as cmpgn\_cmplt\_ts, mssn\_prgrss\_struct\_transactionCount, mssn\_target\_struct, date\_format(regexp\_replace(cmpgn\_start\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as cmpgn\_start\_ts, date\_format(regexp\_replace(cmpgn\_end\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as cmpgn\_end\_ts, cmpgn\_nm, earn\_disc\_amt, date\_format(regexp\_replace(coupn\_exp\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as coupn\_exp\_ts, reward\_pnt\_qty, cmpgn\_tag\_array, cmpgn\_type\_nm, days\_to\_expiry"""  
   
 if self.rptnm == "cr\_burn":  
 self.source\_TB\_ext = "as\_cr\_burn\_ext\_tb"  
 self.column\_str = """singl\_profl\_id, loyalty\_id, burn\_cmpgn\_id, reward\_id, cmpgn\_type\_desc, reward\_status\_desc, date\_format(regexp\_replace(reward\_created\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as reward\_created\_ts, date\_format(regexp\_replace(reward\_exp\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as reward\_exp\_ts, date\_format(regexp\_replace(reward\_redm\_ts,'.000Z',''),"yyyy-MM-dd'T'HH:mm:ss") as reward\_redm\_ts, burn\_disc\_amt"""  
   
 def f\_slice(self):  
   
 # where extract\_date = F.current\_date()  
 dtutl = dtutil()  
 extdt = dtutl.todaystr()  
   
 query\_as = "select " + self.column\_str + " from {}.{} where extract\_date=" + "'" + extdt + "'"  
 df\_as = spark.sql(query\_as.format(self.source\_DB\_ext, self.source\_TB\_ext))  
 df\_as = df\_as.withColumnRenamed("singl\_profl\_id", "named\_user")  
 df\_as = df\_as.select("\*")  
   
 df\_as.show(20)  
   
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\* SPARK JOB Inititated for " + self.rptnm + " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
   
 dtutl = dtutil()  
 out\_folder = dtutl.todaystr()  
 print("out folder is", out\_folder)  
   
 # // bgn Segmenting Files  
   
 # Input Parameters  
 # record\_size\_bytes = self.record\_size\_rpt  
 fseg\_size\_limit\_bytes = 140 \* 1024 \* 1024  
   
 # calculate number of partitions required  
   
 # Writing to Temp Folder, unsegmented csv to get size  
 # /user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_tmp\_files\_as  
 fileoutputstr\_tmp = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_tmp\_files\_as" + "/" + out\_folder  
 print("Writing unsegemented csvs in ... " + fileoutputstr\_tmp)  
 df\_as.write.mode('overwrite').option("compression", "none").csv(fileoutputstr\_tmp)  
   
 file\_path = fileoutputstr\_tmp  
 tot\_file\_size = hdfs\_fs.getContentSummary(hdfs\_path(file\_path)).getLength()  
 print("total file size is ", tot\_file\_size)  
   
 no\_of\_partitions = int((tot\_file\_size) / (fseg\_size\_limit\_bytes)) + 1  
 print("no of partitions is ", no\_of\_partitions)  
   
 if no\_of\_partitions == 0:  
 no\_of\_partitions = 4  
 print("no of partition now set to default of ", no\_of\_partitions)  
   
 df\_as = df\_as.repartition(no\_of\_partitions)  
   
 df\_as.show()  
   
 # Output file segments  
   
 # Set number of records per CSV spreadsheet to (10 million),  
 fseg\_size\_limit\_recs = 10000000  
 print("records limit per file segment is set to ", fseg\_size\_limit\_recs)  
   
 # ~~~ INITIATE writing to HDFS Folder ~~~ #  
   
 # Write to hdfs  
 # /user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_prt\_files\_as/airship\_partfiles\_la\_acct  
 # /user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_prt\_files\_as/airship\_partfiles\_cr\_earn  
 # /user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_prt\_files\_as/airship\_partfiles\_cr\_burn  
   
 # create out\_folder, if not exists.  
   
 fileoutputstr = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_prt\_files\_as/airship\_partfiles\_" + self.rptnm + "/" + out\_folder  
 print("Writing segemented csvs in ... " + fileoutputstr)  
 print("no of partitions is - ", no\_of\_partitions)  
 print("records limit per file segment is set to ", fseg\_size\_limit\_recs)  
 df\_as.write.mode('overwrite').option("compression", "none").option("maxRecordsPerFile", fseg\_size\_limit\_recs).csv(fileoutputstr, header=True)  
   
 # ~~~ END writing to HDFS ~~~ #  
   
  
# end of class  
   
# ~~~ FINISH <jobnm #2:x\_sclicer> ~~~ #  
   
  
# ~~~ START <jobnm #3 :x\_packer > ~~~ #  
   
class x\_packer:  
   
 def \_\_init\_\_(self):  
   
 self.srcfs = hdfs\_fs  
 self.dstfs = hdfs\_fs  
 self.h\_path = hdfs\_path  
   
 def f\_copy(self):  
   
 # \*\*\*\* read from dir prt\_files & copy to dir seg\_files \*\*\*\*  
   
 # COPY from HDFS to HDFS  
   
 srcfs = self.srcfs  
 dstfs = self.dstfs  
 h\_path = self.h\_path  
   
 # read\_from\_dir\_prt = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_prt\_files\_as/airship\_partfiles\_la\_acct/2021-12-06"  
 # write\_to\_dir\_seg ="/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_seg\_files\_as/airship\_segmfiles\_la\_acct/2021-12-06"  
   
 rptnm\_list = ["la\_acct", "cr\_earn", "cr\_burn"]  
 dtutl = dtutil()  
 todaystr = dtutl.todaystr()  
   
 # foreach report directory  
 for rptnm in rptnm\_list:  
   
 # write\_dest\_path  
 hdfs\_dir\_writecopyto = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_seg\_files\_as/airship\_segmfiles\_" + rptnm + "/" + todaystr + "/"  
 print("writing..copy to " + hdfs\_dir\_writecopyto)  
   
 if dstfs.exists(h\_path(hdfs\_dir\_writecopyto)):  
 print("dest directory exists")  
 # delete any existing files in the directory, as they will be written again  
 file\_list\_dir\_prt\_dst = dstfs.listStatus(h\_path(hdfs\_dir\_writecopyto))  
 for file\_del in file\_list\_dir\_prt\_dst:  
 filenm\_del = file\_del.getPath().getName()  
 print("deleting file ... " + filenm\_del + " from " + hdfs\_dir\_writecopyto)  
 dstfs.delete(h\_path(hdfs\_dir\_writecopyto + filenm\_del))  
   
 else:  
 dstfs.mkdirs(h\_path(hdfs\_dir\_writecopyto))  
 # endif  
   
 # read\_source\_path  
 hdfs\_dir\_readcopyfrom = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_prt\_files\_as/airship\_partfiles\_" + rptnm + "/" + todaystr + "/"  
   
 if srcfs.exists(h\_path(hdfs\_dir\_readcopyfrom)):  
 file\_list\_dir\_prt\_src = srcfs.listStatus(h\_path(hdfs\_dir\_readcopyfrom))  
 print("reading..copy from " + hdfs\_dir\_readcopyfrom)  
   
 # foreach file in report source directory  
 for file\_cp in file\_list\_dir\_prt\_src:  
 filenm\_copy = file\_cp.getPath().getName()  
 srcpath = h\_path(hdfs\_dir\_readcopyfrom + filenm\_copy)  
 dstpath = h\_path(hdfs\_dir\_writecopyto + filenm\_copy)  
 fileutil.copy(srcfs, srcpath, dstfs, dstpath, False, jvm\_conf)  
 print(  
 "file copied... " + filenm\_copy + " from " + hdfs\_dir\_readcopyfrom + " to " + hdfs\_dir\_writecopyto)  
 # end for file\_list\_dir\_prt  
   
 # endif  
   
 # end for dir\_list  
   
 def f\_rename(self):  
   
 srcfs = self.srcfs  
 h\_path = self.h\_path  
   
 # read & write to dir\_seg ="/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_seg\_files\_as/airship\_segmfiles\_la\_acct/2021-12-06"  
   
 rptnm\_list = ["la\_acct", "cr\_earn", "cr\_burn"]  
 dtutl = dtutil()  
 todaystr = dtutl.todaystr()  
   
 # foreach report directory  
 for rptnm in rptnm\_list:  
   
 # write\_dest\_path  
 hdfs\_dir\_seg = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_seg\_files\_as/airship\_segmfiles\_" + rptnm + "/" + todaystr + "/"  
   
 # \*\*\*\* read from dir seg\_files & rename all dir seg\_files \*\*\*\*  
 if srcfs.exists(h\_path(hdfs\_dir\_seg)):  
 file\_list\_dir\_seg = srcfs.listStatus(h\_path(hdfs\_dir\_seg))  
   
 counter = 0  
 for file in file\_list\_dir\_seg:  
 filenm\_org = file.getPath().getName()  
 if not (filenm\_org == "\_SUCCESS"):  
 segno = str(counter)  
 filenm\_new = rptnm + "\_seg\_" + segno + "\_as\_" + todaystr + "\_" + "asda" + ".csv"  
 srcfs.rename(h\_path(hdfs\_dir\_seg + filenm\_org), h\_path(hdfs\_dir\_seg + filenm\_new))  
 counter = counter + 1  
 # end for file\_list\_dir\_prt  
   
 # endif  
   
 # end for dir\_list  
   
 def f\_clone\_today(self):  
   
 # \*\*\*\* read from dir prt\_files & copy to dir seg\_files \*\*\*\*  
   
 # COPY from HDFS to HDFS  
   
 srcfs = self.srcfs  
 dstfs = self.dstfs  
 h\_path = self.h\_path  
   
 # read\_from\_dir\_prt = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_seg\_files\_as/airship\_segmfiles\_la\_acct/2021-12-06/"  
 # write\_to\_dir\_snt ="/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_snt\_files\_as/"  
   
 # write\_dest\_path  
 hdfs\_dir\_writecopyto = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_snt\_files\_as/"  
 print("writing..copy to " + hdfs\_dir\_writecopyto)  
   
 # delete any existing files in the directory, before the cloning run  
 file\_list\_dir\_prt\_dst = dstfs.listStatus(h\_path(hdfs\_dir\_writecopyto))  
 for file\_del in file\_list\_dir\_prt\_dst:  
 filenm\_del = file\_del.getPath().getName()  
 print("deleting file ... " + filenm\_del + " from " + hdfs\_dir\_writecopyto)  
 dstfs.delete(h\_path(hdfs\_dir\_writecopyto + filenm\_del))  
   
 rptnm\_list = ["la\_acct", "cr\_earn", "cr\_burn"]  
 dtutl = dtutil()  
 todaystr = dtutl.todaystr()  
   
 # foreach report directory  
 for rptnm in rptnm\_list:  
   
 # read\_source\_path  
 hdfs\_dir\_readcopyfrom = "/user/svc\_uk\_cust\_rdl/marvel\_extracts/today\_seg\_files\_as/airship\_segmfiles\_" + rptnm + "/" + todaystr + "/"  
   
 if srcfs.exists(h\_path(hdfs\_dir\_readcopyfrom)):  
 file\_list\_dir\_seg\_src = srcfs.listStatus(h\_path(hdfs\_dir\_readcopyfrom))  
 print("reading..copy from " + hdfs\_dir\_readcopyfrom)  
   
 # foreach file in report source directory  
 for file\_cp in file\_list\_dir\_seg\_src:  
 filenm\_copy = file\_cp.getPath().getName()  
 srcpath = h\_path(hdfs\_dir\_readcopyfrom + filenm\_copy)  
 dstpath = h\_path(hdfs\_dir\_writecopyto + filenm\_copy)  
 fileutil.copy(srcfs, srcpath, dstfs, dstpath, False, jvm\_conf)  
 print(  
 "file copied... " + filenm\_copy + " from " + hdfs\_dir\_readcopyfrom + " to " + hdfs\_dir\_writecopyto)  
 # end for file\_list\_dir\_prt  
   
 # endif  
   
 # end for dir\_list  
   
  
# end of class  
   
# ~~~ FINISH <jobnm #3:x\_packer> ~~~ #  
   
  
# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*#  
   
  
# instantiate execution for ALL Jobs  
   
  
# Job 1 - writing  
 # rptnm\_list = ["la\_acct", "cr\_earn", "cr\_burn"]  
   
as\_ext\_wrt = x\_writer("la\_acct")  
 as\_ext\_wrt.db\_write()  
   
as\_ext\_wrt = x\_writer("cr\_earn")  
 as\_ext\_wrt.db\_write()  
   
as\_ext\_wrt = x\_writer("cr\_burn")  
 as\_ext\_wrt.db\_write()  
   
# Job 2 - slicing  
 # rptnm\_list = ["la\_acct", "cr\_earn", "cr\_burn"]  
   
as\_ext\_slc = x\_slicer("la\_acct")  
 as\_ext\_slc.f\_slice()  
   
as\_ext\_slc = x\_slicer("cr\_earn")  
 as\_ext\_slc.f\_slice()  
   
as\_ext\_slc = x\_slicer("cr\_burn")  
 as\_ext\_slc.f\_slice()  
   
# Job 3 - packing  
 as\_ext\_pck = x\_packer()  
 as\_ext\_pck.f\_copy()  
 as\_ext\_pck.f\_rename()  
 as\_ext\_pck.f\_clone\_today()  
   
# Job 4 - sending  
 # as\_ext\_snd = x\_sender()  
 # as\_ext\_snd.f\_send()