WRAPUP: Work Smarter

Time To Bring In Z Open Automation Utilities

12 steps 90 minutes

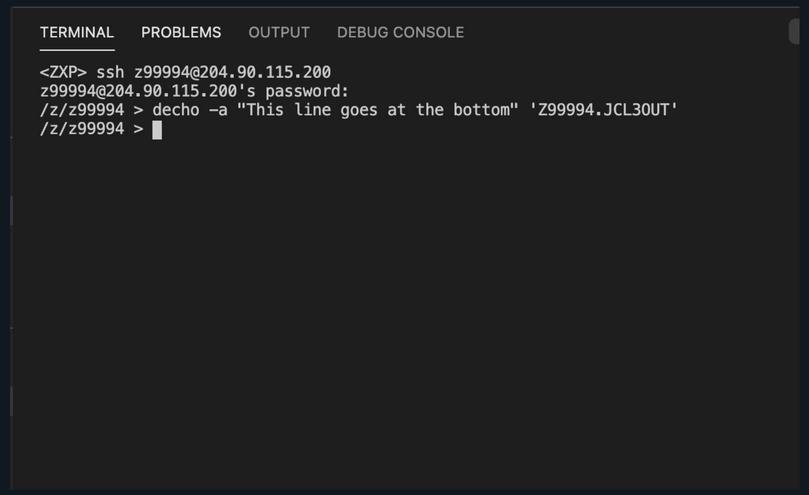
THE CHALLENGE

Now that you know some of the core z/OS fundamentals, and you've also explored a bit into scripting and Python, let's bring it all together with something called Z Open Automation Utilities, or to save some space from now on, ZOAU.

We'll combine the core Z skills we've picked up from learning about data sets and JCL with the scripting abilities found in Python to automate some tasks a System Programmer might perform on a day-to-day basis.

BEFORE YOU BEGIN

At this point, you know how to work with data sets, submit jobs, and navigate around USS. You'll put all that to use here and wrap up your fundamental understanding of z/OS.



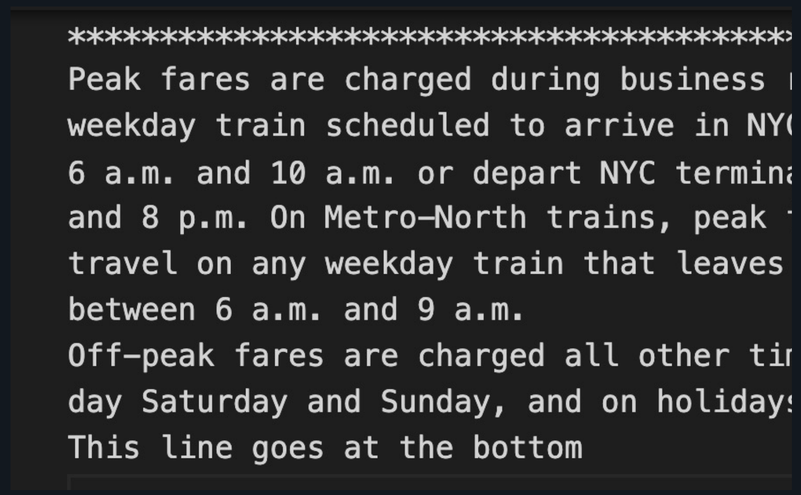
1. DECHO... DECHO... DECHO

You'll need a sequential data set for this challenge. If you don't still have your JCL3OUT sequential data set, make one by right-clicking on your connection profile in Data Sets and selecting "Create New Data Set" and following the prompts.

All set? Alright, SSH into the system again so you get a shell and then enter the correct version of this command, substituting in your own userid and the data set you want to use.

decho -a "This line goes at the bottom" 'Z99994.JCL3OUT'

This may take a few seconds to fully execute, so be patient.

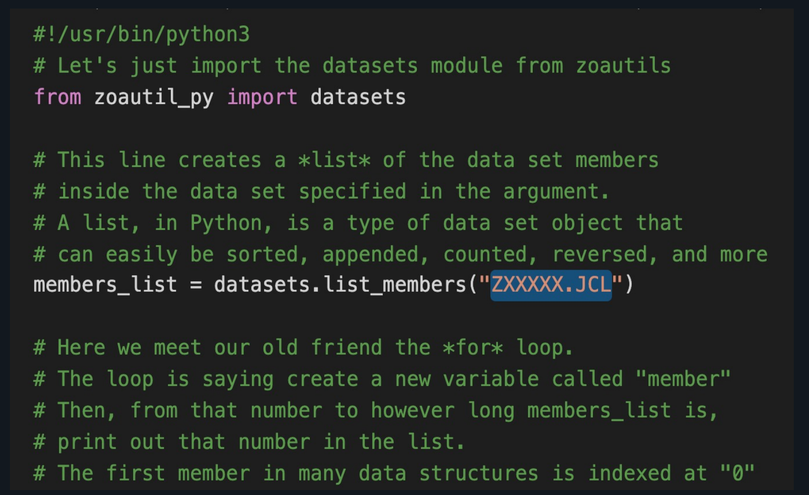


2. HELLO DOWN THERE

Open up the data set you used in the command in VS Code. You may need to right-click on it and select "Pull From Mainframe" if it's already open in order to get the absolute latest version.

You should see the line you just *decho*'d appended to the bottom of the data set. Neat trick, though probably not easier than just opening it up and typing it in there manually.

The real power of these tools comes from being able to integrate z/OS actions into new and existing Python programs and shell scripts.



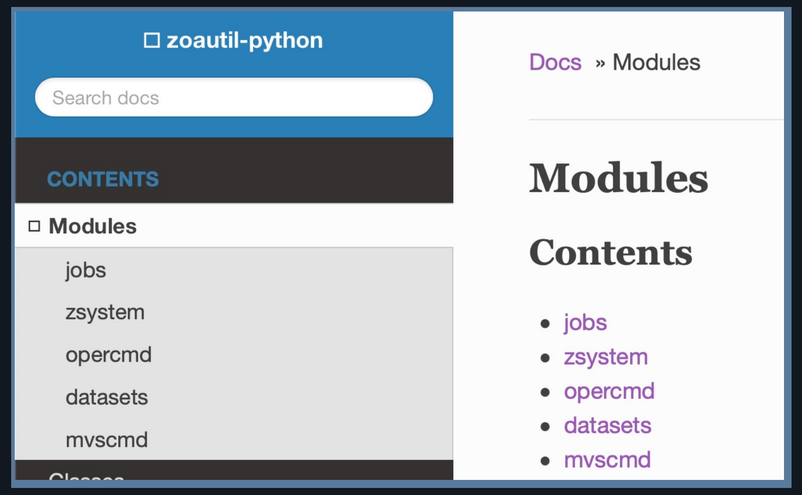
```
#!/usr/bin/python3
# Let's just import the datasets module from zoautils
from zoautil_py import datasets

# This line creates a *list* of the data set members
# inside the data set specified in the argument.
# A list, in Python, is a type of data set object that
# can easily be sorted, appended, counted, reversed, and more
members_list = datasets.list_members("ZXXXXX.JCL")

# Here we meet our old friend the *for* loop.
# The loop is saying create a new variable called "member"
# Then, from that number to however long members_list is,
# print out that number in the list.
# The first member in many data structures is indexed at "0"
```

3. ALL SET WITH DATA SETS

Look for **dslist.py** in /z/public/ and copy it over to your own home directory. You can take a quick peek at the code to figure out what it's doing.

It's a Python script (if you didn't guess by the .py suffix) and you can see it gets a listing of everything in a particular data set, then uses a *for* loop to print out each member name in it (at least, it will, once you go in and edit to point to one of your data sets on line number 9). Make that change now. Try your JCL or OUTPUT data set.

Nice and simple, right?



4. THERE'S A PYTHON IN MY Z

Writing a Python script just to list what's in a data set might seem like overkill, and in this case... yes, it is. However, we want to show you, in a very simple program, just how it can be done. And now that you see it, maybe you have some ideas.

Now that we know we can perform core z/OS tasks within Python code. This example only involved data sets. You can also work with jobs, the system itself, and two types of commands. Take a look at the Python APIs page and read more about these modules HERE.

```
#!/usr/bin/python3
# Let's just import the datasets module from zoautils
from zoautil_py import datasets, jobs, zsystem
import sys

#Prompt for data set name:
dsname = input("Enter the Sequential Data Set name:")

#if it exists, say we foudn it, and we'll use it. Otherw
if (datasets.exists(dsname) == True):
    print("Data set found! We will use it")
else:
    create_new = input("Data set not found. Should we cr
    if (create_new.upper() == "Y"):
```

5. INSERTING SOME LOGIC

Next, we'll work with a script that does a bit more, including creating a new sequential data set, gathering some data, and then writing that data out to our newly-created sequential data set.

We'll get you started on a couple of these functions, and your job will be to get the rest of them working smoothly.

If it's not in your home directory already, copy over **members.py** from /z/public and take a look at the source.

```
# These are comments
# They are here to help
```

6. GET YOUR HACK ON

At this point, you might be seeing a lot of code and getting nervous. Here's the deal... even if you don't want to be a programmer, you'll have to spend some time looking at code and making small adjustments to other people's code. That is the very spirit of hacking.

A lot of coding involves figuring out where to start, pulling together all of the resources you need, and creating something from scratch. Hacking is taking something that does something and making small changes to add a feature, fix problems, or just leave your own little mark on it.

Whenever there's code involved, we've left LOTS of comments to help you figure out what's going on, and know that at least for this challenge, you'll never need to write more than a few lines of new code. Figuring out the flow and structure of what's already there will probably be the hardest thing.

You got this!

WHY LEARN ANOTHER WAY? BECAUSE IT'S GOOD TO HAVE OPTIONS.

If you started working on z/OS years ago, chances are you're very comfortable with JCL and the many methods already built into the operating system for getting things done.

However, if you're used to working with Linux and Python, you may want to keep writing code the same way, while still having access to the core functionality of z/OS. That's where the ZOAU library modules come in handy.

We learn new methods in addition to what we already know so we have more options, and more opportunity to make efficient choices. These exercises are here to give you a few ideas of what is possible when you combine the scripting and automation ability of Python with the backend capabilities of z/OS.

```
#!/usr/bin/python3
# Let's just import the datasets module from zoaut
from zoautil_py import datasets, jobs, zsystem
import sys

#Prompt for data set name:
dsname = input("Enter the Sequential Data Set name

#if it exists, say we foudn it, and we'll use it.
if (datasets.exists(dsname) == True):
    print("Data set found! We will use it")
else:
```

7. LOADING MODULES

Look at line #3 of **members.py**. We're choosing to import three modules from the **zoautil.py** library; datasets, jobs, and zsystem. Each of these modules provides some set of functions which we can now use in our code.

You can read all about the modules and what they do here: https://www.ibm.com/docs/api/v1/content/SSKFYE_1.1.1/python_doc_zoautil/index.html?view=embed

```
#Prompt for data set name:
dsname = input("Enter the Sequential Data Set name:")

#if it exists, say we foudn it, and we'll use it. Otherwi
if (datasets.exists(dsname) == True):
    print("Data set found! We will use it")
else:
    create_new = input("Data set not found. Should we cre
    if (create_new.upper() == "Y"):
        # User wants to create a file
        # This is the part where we create a new data set
        datasets.create(dsname,type="???",primary_space="
    else: sys.exit("Without a data set name, we cannot co
```

8. GET THAT DATA SET

Lines 6-18 ask the user for a sequential data set name and assigns that value to the variable *dsname*.

If the data set already exists, we'll use that and continue on. If the data set does not exist, we can create that for the user (which we'll do in the next step).
If the data set does NOT exist, and the user doesn't want to create it, then there isn't much sense in continuing the rest of the code, so we use *sys.exit* to quit the program.

So far, so good? Alright, let's start hacking some code.

```
input("Data set not found. Should we crea
v.upper() == "Y"):
nts to create a file
 the part where we create a new data set
create(dsname,type="???",primary_space="1
t("Without a data set name, we cannot con

orrect line of code from the 4 lines belo
the system's linklist representation and
 the variable 'linklist_output'
```

9. IN THE RIGHT SEQUENCE

For this challenge, we'll be using this script (**members.py**) to gather data and write it out to a sequential data set.

Line #17 is using the *create* function from the *datasets* module to attempt to create this sequential data set. But as it is right now, it will not work until you make one small adjustment. Take a look at the *datasets.create* function and figure out what needs to replace the ??? in order to create a *sequential* data set.

Use ZXXXXX.COMPLETE as the sequential data set when prompted by the script. That's where we'll be looking to validate your work, in addition to copy of **members.py**. (Remember to use your userid instead of ZXXXXX!).

Hint: A sequential data set is a *type* of data set. Other types of data sets are KSDS, PDS, ESDS, but in this case, the type of data set we want to create is a sequential data set.

KSDS? ESDS? I THOUGHT IT WAS JUST A PARTITIONED AND SEQUENTIAL?

Sometimes a data set is a little more than a simple file; a place to hold some data that you want to read or process from top to bottom. Data set members and sequential data sets work great for these cases.

However, sometimes applications need fast access to a specific record, and they need a better way of getting to that record than reading the whole data set top to bottom. In these situations, we can use a key Sequenced Data Set (KSDS). Entry Sequenced Data Set (ESDS) or Relative Record Data Set (RRDS). These are all examples of Virtual Storage Access Method (VSAM) data sets, and we'll revisit them later.

Having options when it comes to data access means we can choose the best, fastest, most efficient way of working with all those bits and bytes. Understanding the options and how to make it all work makes you a valuable z/OS professional, and employers definitely like that sort of thing.

```
# Uncomment the correct line of code from the 4 lines
# which will get the system's linklist representation
# its contents to the variable 'linklist_output'
# https://www.ibm.com/docs/en/zoau/1.1.1?topic=SSKFYE_
#linklist_output = zsystem.get_linklist()
#linklist_output = zsystem.list_linklist()
#linklist_output = zsystem.link_linklist()
#linklist_data = zsystem.list_linklist()

#Format the output so each member is on its own line
linklist_output = str(linklist_output).replace(',','\n
print(linklist_output)

# Write the value of linklist_info into our sequential
```

10. GRAB THE LINKLIST

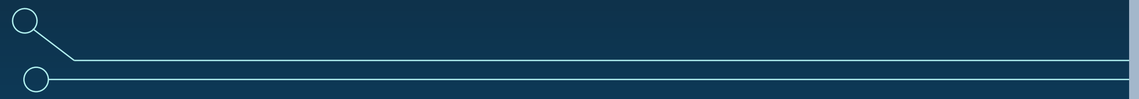
Remember how we imported some modules at the beginning of this script? When the z/OS operating system starts up, it kind of does the same thing, loading modules and libraries full of the types of capabilities the user might need to use.

The full list of loaded libraries is known as the linklist, and it's what lets a user just type out a single command instead of having to reference it by its full path every time. Very handy!

Anyway, knowing what the full linklist looks like is kind of important, right within the *zsystem* module of zoautil.py

Look at lines 20-27. As you can see, there are four lines of code in there (and a lot of comments in case you feel like reading them...which you probably should).

Your task is to identity and uncomment the one line of code (from 24-27) that is the correct line that will grab the linklist representation and assign it to the *linklist_output* variable.



```
#linklist_data = zsystem.list_linklist()

#This is just here to show the value of linklist_output
print(linklist_output)

# Write the value of linklist_info into our sequential
# This is the data set we created back on line xx
datasets.write(linklist_output,dsname,append=False)

# If everything looks good, run the JCL for this challe
# For bonus points (bonus points may not actually exist
# see if you can submit the JCL through this script.
```

11. LAST STOP: WRITING OUT

So far, we've created (or re-used) a sequential data set and gathered some data. Let's write that data out into the data set at end of our script.

All the information is there, but *something* is wrong and you'll need to take a look at the datasets.write method to figure out what it is.

Once you've got it all straightened out and in order, give it a quick run and make sure the script works for you.

```
['VENDOR.LINKLIB'
 'SYS1.MIGLIB'
 'SYS1.CSSLIB'
 'SYS1.SIEALNKE'
 'SYS1.SIEAMIGE'
 'SVTSC.LINKLIB'
 'LVL0.LOADLIB'
 'LVL0.LINKLIB'
 'SYS1.LINKLIB'
 'SYS1.CMDLIB'
 'SYS1.SHASLNKE'
 'SYS1.SHASMIG'
 'ISF.SISFLOAD'
 'ISF.SISFLINK'
```

12. CHECK IT. FINISH IT.

If you completed all of the steps, you should have a ZXXXXX.COMPLETE sequential data set that contains a readout of the system's linklist. Refer to the screenshot above if you need some clarification.

Find and submit CHKAUTO to mark this one complete. You can do that the way you've been doing it up until now, or...I bet there's a function for submitting JCL that might work nicely.

Either way, congratulations on completing the fundamentals challenges! You're all set to go bigger and further.

NICE JOB! LET'S RECAP

You scripted a series of deep fundamental z/OS actions right from a Python script! We kept it fairly simple in this example, but now that you know how to connect the dots, you should be able to create any number of new utilities that may be helpful in processing text, checking job output, verifying system changes, and working with data sets.

You knew how to do most of this before, and now you know another way. Like we said, more options are good to have.

NEXT UP...

You've completed the fundamental challenges! You've got what it takes to move on to the advanced challenges. By now, you probably have some idea of what you're interested in, and you'll probably find it in the next series of available challenges.