Automating AWS With Python and Boto3

1/26/2017 - COURSE:
AWS CERTIFIED DEVELOPER - ASSOCIATE LEVEL
INSTRUCTOR APPROVED

TABLE OF CONTENTS

1. INTRODUCTION
2. REQUIREMENTS
3. GETTING CONFIGURED
4. PYTHON AND PIP
5. AWS CLI TOOL AND BOTO3
6. USERS, PERMISSIONS, AND CREDENTIALS
7. SCRIPTING EC2
8. LIST INSTANCES
9. CREATE AN INSTANCE
10. GETTING AN AMI ID
11. TERMINATE AN INSTANCE
12. SCRIPTING S3
13. LIST BUCKETS AND THEIR CONTENTS
14. CREATE A BUCKET
15. PUT A FILE INTO A BUCKET
16. DELETE BUCKET CONTENTS
17. DELETE A BUCKET
18. SCRIPTING RDS
19. LIST DB INSTANCES
20. CREATE A DB INSTANCE
21. DELETE A DB INSTANCE
22. WRAPPING UP AND NEXT STEPS

INTRODUCTION

In this tutorial, we'll take a look at using Python scripts to interact with infrastructure provided by Amazon Web Services (AWS). You'll learn to configure a workstation with Python and the Boto3 library. Then, you'll learn how to programmatically create and manipulate:

- Virtual machines in Elastic Compute Cloud (EC2)
- Buckets and files in Simple Storage Service (S3)
- Databases in Relational Database Service (RDS)

REQUIREMENTS

Before we get started, there are a few things that you'll need to put in place:

- An AWS account with admin or power user privileges. Since we'll be creating, modifying, and deleting things in this exercise, the account should be a sandbox account that does not have access to production VMs, files, or databases.
- Access to a Linux shell environment with an active internet connection.
- Some experience working with Python and the Bash command line interface.

GETTING CONFIGURED

Let's get our workstation configured with Python, Boto3, and the AWS CLI tool. While the focus of this tutorial is on using Python, we will need the AWS CLI tool for setting up a few things.

Once we're set up with our tools on the command line, we'll go to the AWS console to set up a user and give permissions to access the services we need to interact with.

PYTHON AND PIP

First, check to see if Python is already installed. You can do this by typing *which python* in your shell. If Python is installed, the response will be the path to the Python executable. If Python is not installed, go to the Python.org website for information on downloading and installing Python for your particular operating system.

We will be using Python 2.7.10 for this tutorial. Check your version of Python by typing *python -V*. Your install should work fine as long as the version is 2.6 or greater.

The next thing we'll need is pip, the Python package manager. We'll use pip to install the Boto3 library and the AWS CLI tool. You can check for pip by typing *which pip*. If pip is installed, the response will be the path to the pip executable. If pip is not installed, follow the instructions at pip.pypa.io to get pip installed on your system.

Check your version of pip by typing "pip -V". Your version of pip should be 9.0.1 or newer.

Now, with Python and pip installed, we can install the packages needed for our scripts to access AWS.

AWS CLI TOOL AND BOTO3

Using the pip command, install the AWS CLI and Boto3:

```
pip install awscli boto3 -U --ignore-installed six
```

Please Note: This command may need to be run with sudo to allow for installation with elevated privileges. The -U option will upgrade any packages if they are already installed. On some systems, there may be issues with a package named "six"; using the "--ignore-installed six" option can work around those issues.

We can confirm the packages are installed by checking the version of the AWS CLI tool and loading the boto3 library.

Run the command "aws --version" and something similar to the following should be reported:

aws-cli/1.11.34 Python/2.7.10 Darwin/15.6.0 botocore/1.4.91

Finally, run the following to check boto3: *python -c "import boto3"*. If nothing is reported, all is well. If there are any error messages, review the setup for anything you might have missed.

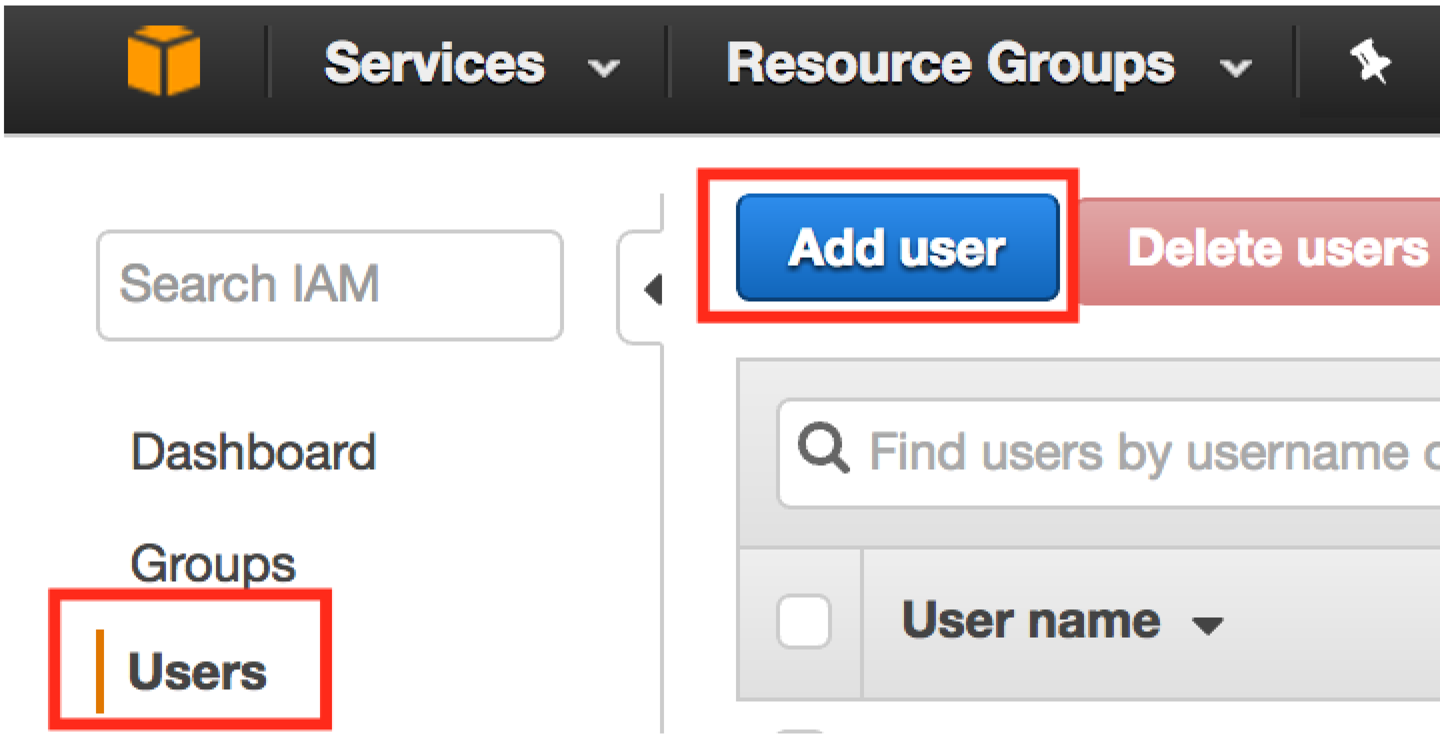
At this point, we're ready for the last bit of configuration before we start scripting.

USERS, PERMISSIONS, AND CREDENTIALS

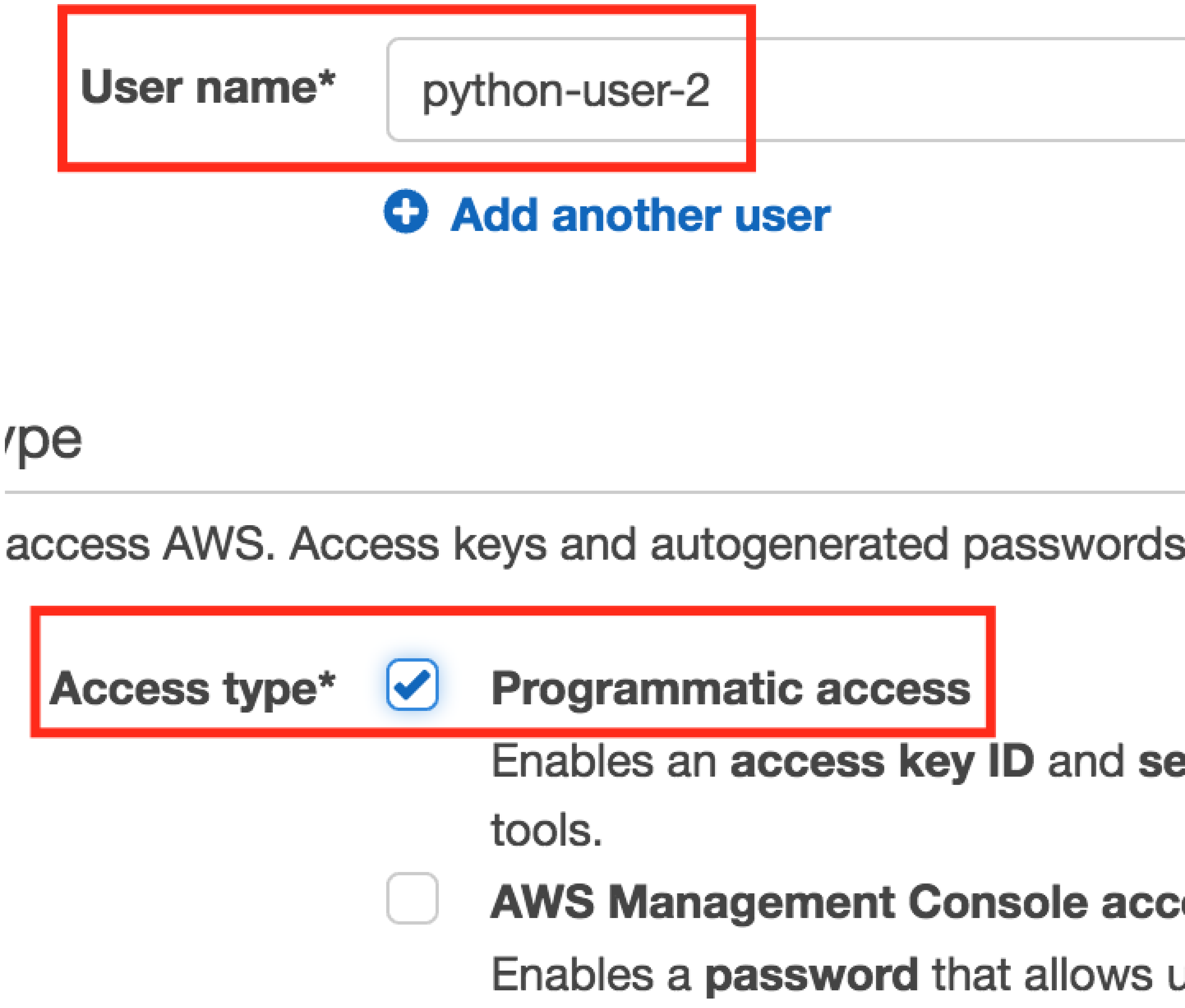
Before we can get up and running on the command line, we need to go to AWS via the web console to create a user, give the user permissions to interact with specific services, and get credentials to identify that user.

Open your browser and navigate to the AWS login page. Typically, this is https://console.aws.amazon.com/console/home.

Once you are logged into the console, navigate to the Identity and Access Management (IAM) console. Select "Users" -> "Add user."



On the “Add user” page, give the user a name and select “Programmatic access." Then click “Next: Permissions." In this example, I’ve named the user “python-user-2." Note that your user name should not have any spaces or special characters.



User name* python-user-2

ʼpe

access AWS. Access keys and autogenerated passwords

Access type* [✓] **Programmatic access**

Enables an **access key ID** and **se**

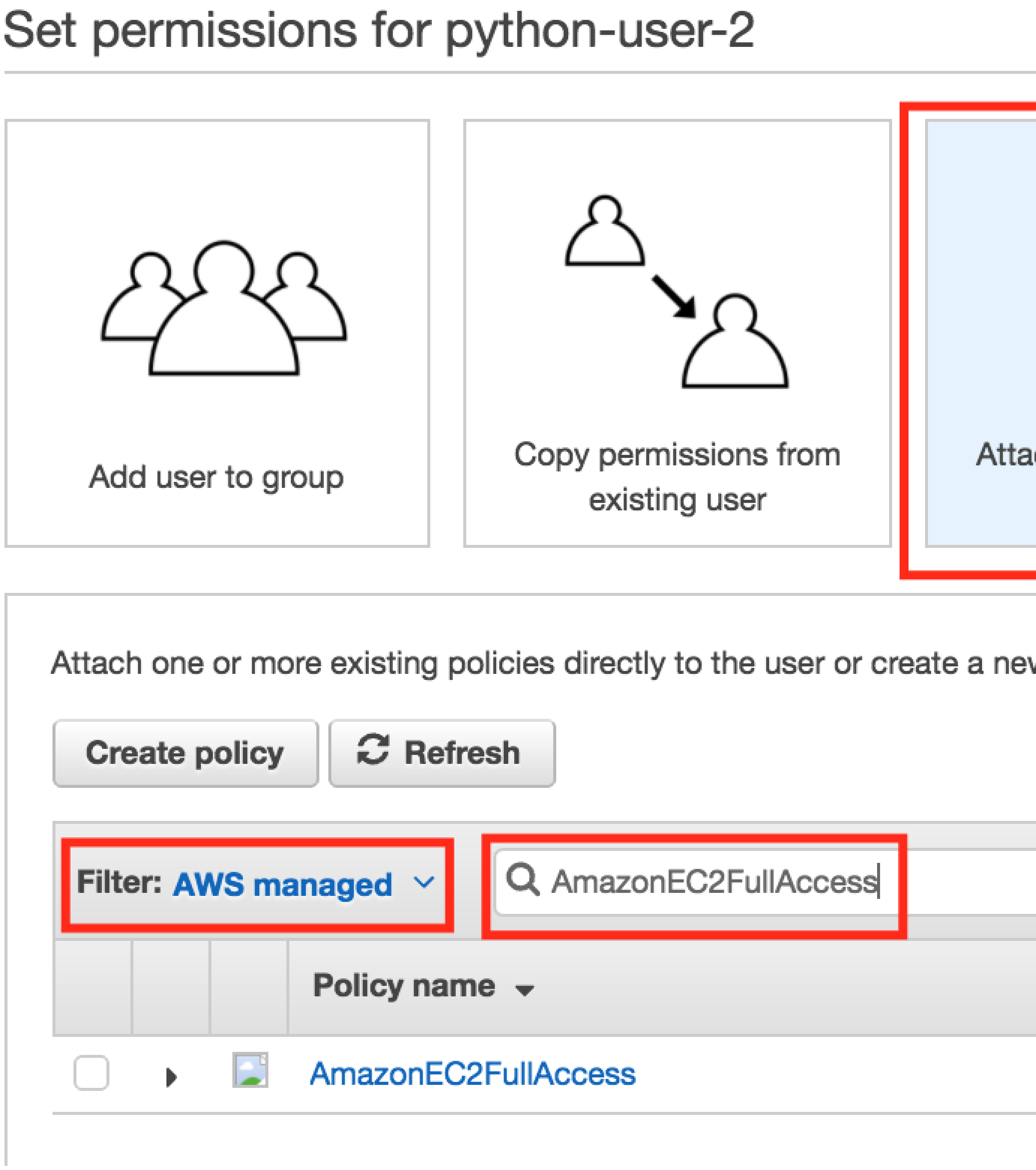
tools.

[ ] **AWS Management Console acc**

Enables a **password** that allows u

On the “Permissions” page, we will set permissions for our user by attaching existing policies directly to our user. Click “Attach existing policies directly." Next to “Filter”, select “AWS managed." Now search for

AmazonEC2FullAccess. After entering the search term, click the box next to the listing for “AmazonEC2FullAccess." Repeat this step for S3 and RDS, searching for and selecting AmazonS3FullAccess and AmazonRDSFullAccess. Once you’ve selected all three, click “Next: Review."


Set permissions for python-user-2

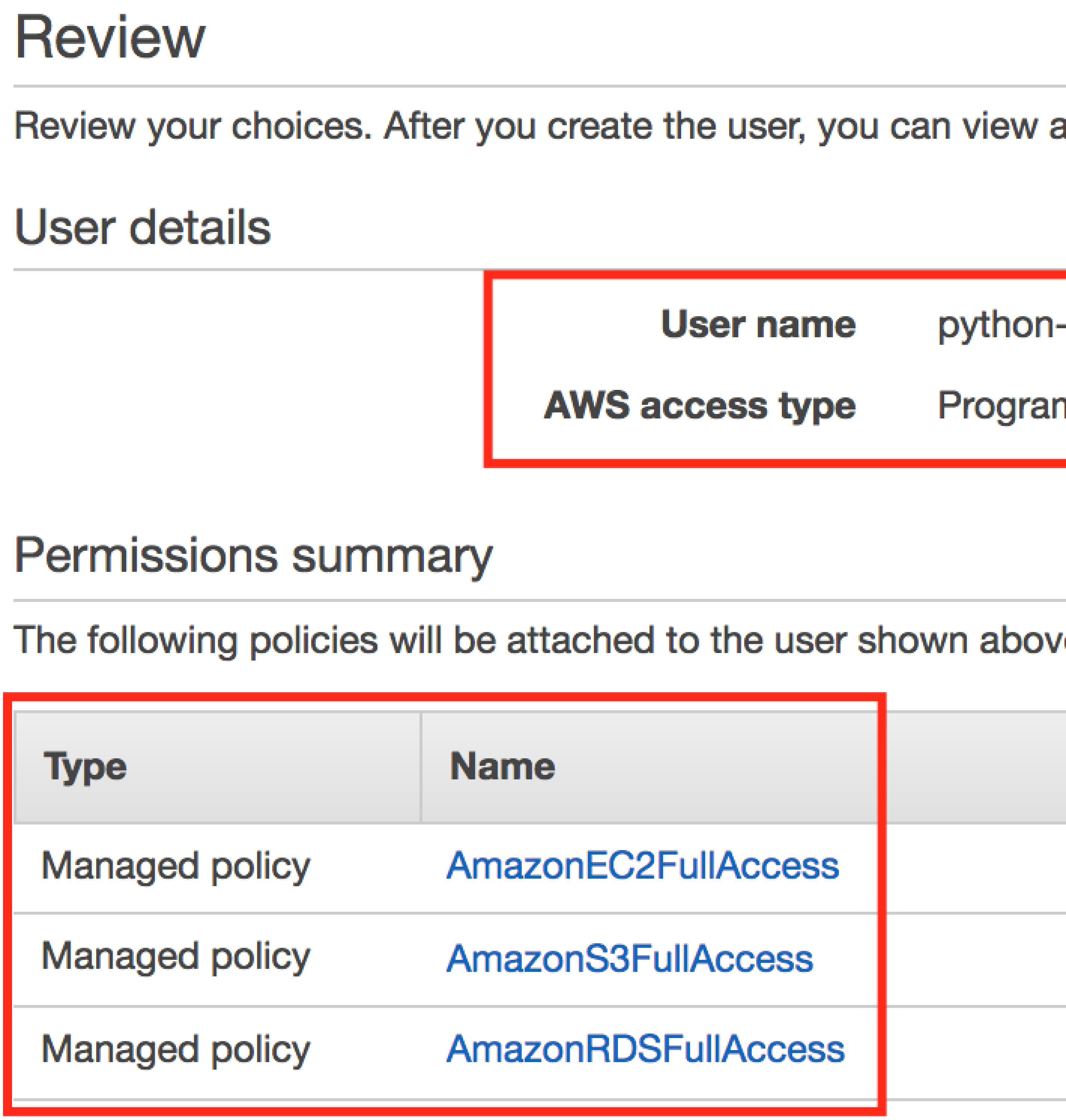
Attach one or more existing policies directly to the user or create a nev

Filter: AWS managed

AmazonEC2FullAccess

Policy name
AmazonEC2FullAccess

On the review screen, check your user name, AWS access type, and permissions summary. It should be similar to the image below. If you need to fix anything, click the “Previous” button to go back to prior screens and make changes. If everything looks good, click “Create user."



Review

Review your choices. After you create the user, you can view a

User details

User name	python-
AWS access type	Progran

Permissions summary

The following policies will be attached to the user shown abov

Type	Name
Managed policy	AmazonEC2FullAccess
Managed policy	AmazonS3FullAccess
Managed policy	AmazonRDSFullAccess

On the final user creation screen, you'll be presented with the user's access key ID and secret access key. Click the "Download .csv" button to save a text file with these credentials or click the “Show” link next to the secret access key. IMPORTANT: Save the file or make a note of the credentials in a safe place as this is the only time that they are easily captured. Protect these credentials like you would protect a username and password!

Now that we have a user and credentials, we can finally configure the scripting environment with the AWS CLI tool.

Back in the terminal, enter *aws configure*. You’ll be prompted for the AWS access key ID, AWS secret access key, default region name, and default output format. Using the credentials from the user creation step, enter the access key ID and secret access key.

For the default region name, enter the region that suits your needs. The region you enter will determine the location where any resources created by your script will be located. You can find a list of regions in the AWS documentation. In the example below, I’m using us-west-2.

Options for the default output format are text, json, and table. Enter “text” for now.

```
AWS Access Key ID [None]: AKIAJFUD42GXIN4SQRKA

AWS Secret Access Key [None]: LLL1tjMJpRNsCq23AXVtZXLJhvYkjHeDf4UO9zzz

Default region name [None]: us-west-2

Default output format [None]: text
```

Now that your environment is all configured, let’s run a quick test with the AWS CLI tool before moving on. In the shell, enter:

```
aws ec2 describe-instances
```

If you already have instances running, you’ll see the details of those instances. If not, you should see an empty response. If you see any errors, walk through the previous steps to see if anything was overlooked or entered incorrectly, particularly the access key ID and secret access key.

Wow! That was a lot to set up but we’re finally ready to do some scripting. Let’s get started with some basic scripts that work with EC2, S3, and RDS.

SCRIPTING EC2

The Elastic Compute Cloud (EC2) is a service for managing virtual machines running in AWS. Let’s see how we can use Python and the boto3 library with EC2.

LIST INSTANCES

For our first script, let’s list the instances we have running in EC2. We can get this information with just a few short lines of code.

First, we’ll import the boto3 library. Using the library, we’ll create an EC2 resource. This is like a handle to the EC2 console that we can use in our script. Finally, we’ll use the EC2 resource to get all of the instances and then print their instance ID and state. Here’s what the script looks like:

```
#!/usr/bin/env python

import boto3

ec2 = boto3.resource('ec2')

for instance in ec2.instances.all():

    print instance.id, instance.state
```

Save the lines above into a file named list_instances.py and change the mode to executable. That will allow you to run the script directly from the command line. Also note that you’ll need to edit and chmod +x the remaining scripts to get them running as well. In this case, the procedure looks like this:

```
$ vi list_instances.py

$ chmod +x list_instances.py

$ ./list_instances.py
```

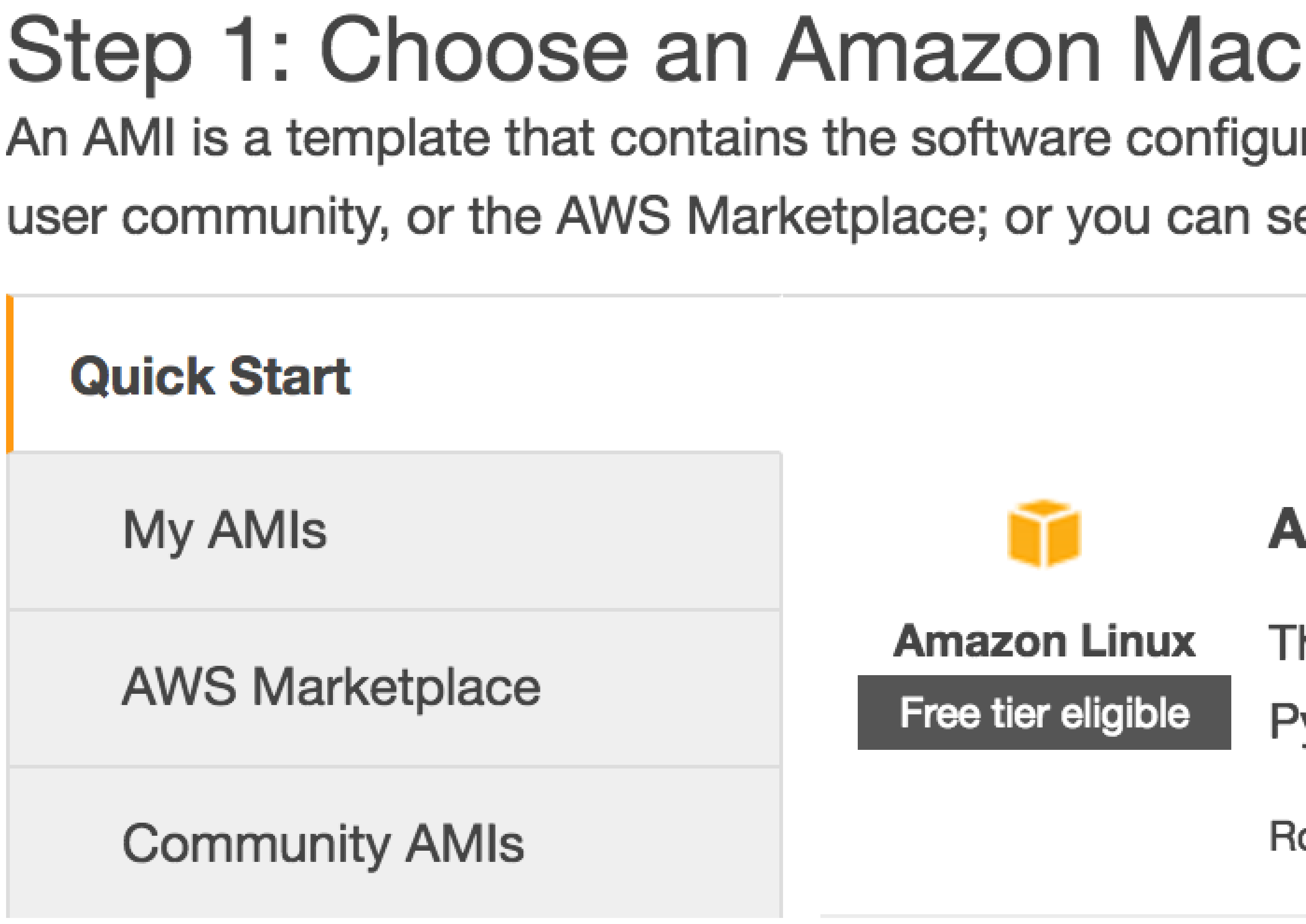
If you haven’t created any instances, running this script won’t produce any output. So let’s fix that by moving on to the the next step and creating some instances.

CREATE AN INSTANCE

One of the key pieces of information we need for scripting EC2 is an Amazon Machine Image (AMI) ID. This will let us tell our script what type of EC2 instance to create. While getting an AMI ID can be done programmatically, that's an advanced topic beyond the scope of this tutorial. For now, let’s go back to the AWS console and get an ID from there.

In the AWS console, go the the EC2 service and click the “Launch Instance” button. On the next screen, you’re presented with a list of AMIs you can use to create instances. Let’s focus on the Amazon Linux AMI at the very top of the list. Make a note of the AMI ID to the right of the name. In this example, its “ami-1e299d7e." That’s the value we need for our script. Note that AMI IDs differ across regions and are updated often so the latest ID for the Amazon Linux AMI may be different for you.

GETTING AN AMI ID


Step 1: Choose an Amazon Mac
An AMI is a template that contains the software configu
user community, or the AWS Marketplace; or you can se
Quick Start
My AMIs
AWS Marketplace
Community AMIs
Amazon Linux
Free tier eligible
A
Th
Py
Ro

Now with the AMI ID, we can complete our script. Following the pattern from the previous script, we’ll import the boto3 library and use it to create an EC2 resource. Then we’ll call the create_instances() function, passing in the image ID, max and min counts, and the instance type. We can capture the output of the function call which is an instance object. For reference, we can print the instance's ID.

```
#!/usr/bin/env python

import boto3

ec2 = boto3.resource('ec2')

instance = ec2.create_instances(

    ImageId='ami-1e299d7e',

    MinCount=1,

    MaxCount=1,

    InstanceType='t2.micro')

print instance[0].id
```

While the command will finish quickly, it will take some time for the instance to be created. Run the *list_instances.py* script several times to see the state of the instance change from pending to running.

TERMINATE AN INSTANCE

Now that we can programmatically create and list instances, we also need a method to terminate them.

For this script, we'll follow the same pattern as before with importing the boto3 library and creating an EC2 resource. But we'll also take one parameter: the ID of the instance to be terminated. To keep things simple, we'll consider any argument to the script to be an instance ID. We'll use that ID to get a connection to the instance from the EC2 resource and then call the terminate() function on that instance. Finally, we print the response from the terminate function. Here's what the script looks like:

```
#!/usr/bin/env python

import sys

import boto3

ec2 = boto3.resource('ec2')

for instance_id in sys.argv[1:]:

    instance = ec2.Instance(instance_id)

    response = instance.terminate()

    print response
```

Run the *list_instances.py* script to see what instances are available. Note one of the instance IDs to use as input to the *terminate_instances.py* script. After running the terminate script, we can run the list instances script to confirm the selected instance was terminated. That process looks something like this:

```
$ ./list_instances.py

i-0c34e5ec790618146 {u'Code': 16, u'Name': 'running'}

$ ./terminate_instances.py i-0c34e5ec790618146

{u'TerminatingInstances': [{u'InstanceId': 'i-0c34e5ec790618146', u'CurrentState':
{u'Code': 32, u'Name': 'shutting-down'}, u'PreviousState': {u'Code': 16, u'Name':
'running'}}], 'ResponseMetadata': {'RetryAttempts': 0, 'HTTPStatusCode': 200,
'RequestId': '55c3eb37-a8a7-4e83-945d-5c23358ac4e6', 'HTTPHeaders': {'transfer-
encoding': 'chunked', 'vary': 'Accept-Encoding', 'server': 'AmazonEC2', 'content-
type': 'text/xml;charset=UTF-8', 'date': 'Sun, 01 Jan 2017 00:07:20 GMT'}}}


$ ./list_instances.py

i-0c34e5ec790618146 {u'Code': 48, u'Name': 'terminated'}
```

SCRIPTING S3

The AWS Simple Storage Service (S3) provides object storage similar to a file system. Folders are represented as buckets and the contents of the buckets are known as keys. Of course, all of these objects can be managed with Python and the boto3 library.

LIST BUCKETS AND THEIR CONTENTS

Our first S3 script will let us see what buckets currently exist in our account and any keys inside those buckets.

Of course, we'll import the boto3 library. Then we can create an S3 resource. Remember, this gives us a handle to all of the functions provided by the S3 console. We can then use the resource to iterate over all buckets. For each bucket, we print the name of the bucket and then iterate over all the objects inside that bucket. For each object, we print the object's key or essentially the object's name. The code looks like this:

```
#!/usr/bin/env python

import boto3

s3 = boto3.resource('s3')

for bucket in s3.buckets.all():

    print bucket.name
    print "---"
    for item in bucket.objects.all():
        print "\t%s" % item.key
```

If you don’t have any buckets when you run this script, you won’t see any output. Let’s create a bucket or two and then upload some files into them.

CREATE A BUCKET

In our bucket creation script, let's import the boto3 library (and the sys library too for command line arguments) and create an S3 resource. We’ll consider each command line argument as a bucket name and then, for each argument, create a bucket with that name.

We can make our scripts a bit more robust by using Python’s *try* and *except* features. If we wrap our call to the *create_bucket()* function in a *try:* block, we can catch any errors that might occur. If our bucket creation goes well, we simply print the response. If an error is encountered, we can print the error message and exit gracefully. Here’s what that script looks like:

```
#!/usr/bin/env python
import sys
import boto3
s3 = boto3.resource("s3")
for bucket_name in sys.argv[1:]:
    try:
        response = s3.create_bucket(Bucket=bucket_name)
        print response
    except Exception as error:
        print error
```

Creating a bucket is easy but comes with some rules and restrictions. To get the complete run-down, read the Bucket Restrictions and Limitations section in the S3 documentation. The two rules that needs to be emphasized for this example are 1) bucket names must be globally unique and 2) bucket names must follow DNS naming conventions.

Basically, when choosing a bucket name, pick one that you are sure hasn’t been used before and only use lowercase letters, numbers, and hyphens.

Because simple bucket names like “my_bucket” are usually not available, a good way to get a unique bucket name is to use a name, a number, and the date. For example:

```
$ ./create_bucket.py projectx-bucket1-$(date +%F-%s)
s3.Bucket(name='projectx-bucket1-2017-01-01-1483305884')
```

Now we can run the list_buckets.py script again to see the buckets we created.

```
$ ./list_buckets.py
projectx-bucket1-2017-01-01-1483305884
```

OK! Our buckets are created but they’re empty. Let’s put some files into these buckets.

PUT A FILE INTO A BUCKET

Similar to our bucket creation script, we start the put script by importing the sys and boto3 libraries and then creating an S3 resource. Now we need to capture the name of the bucket we’re putting the file into and the name of the file as well. We’ll consider the first argument to be the bucket name and the second argument to be the file name.

To keep with robust scripting, we’ll wrap the call to the put() function in a try; block and print the response if all goes well. If anything fails, we’ll print the error message. That script comes together like this:

```
#!/usr/bin/env python
import sys
import boto3
s3 = boto3.resource("s3")
bucket_name = sys.argv[1]
object_name = sys.argv[2]
try:
    response = s3.Object(bucket_name, object_name).put(Body=open(object_name, 'rb'))
    print response
except Exception as error:
    print error
```

For testing, we can create some empty files and then use the *put_bucket.py* script to upload each file into our target bucket.

```
$ touch file{1,2,3,4}.txt

$ ./put_bucket.py projectx-bucket1-2017-01-01-1483305884 file1.txt

{u'ETag': '"d41d8cd98f00b204e9800998ecf8427e"', 'ResponseMetadata':
{'HTTPStatusCode': 200, 'RetryAttempts': 0, 'HostId':
'Zf7Ti20qpyBr/ssSOYVDzc3501dDuSApWJzqnq/bhfGzPWbEqnyI7I2pLoMbRbtl2ltMlNXg5Zo=',
'RequestId': '1EDAE2B1F66C693D', 'HTTPHeaders': {'content-length': '0', 'x-amz-id-2':
'Zf7Ti20qpyBr/ssSOYVDzc3501dDuSApWJzqnq/bhfGzPWbEqnyI7I2pLoMbRbtl2ltMlNXg5Zo=',
'server': 'AmazonS3', 'x-amz-request-id': '1EDAE2B1F66C693D', 'etag':
'"d41d8cd98f00b204e9800998ecf8427e"', 'date': 'Sun, 01 Jan 2017 21:45:28 GMT'}}}


$ ./put_bucket.py projectx-bucket1-2017-01-01-1483305884 file2.txt

...

$ ./put_bucket.py projectx-bucket1-2017-01-01-1483305884 file3.txt

...

$ ./put_bucket.py projectx-bucket1-2017-01-01-1483305884 file4.txt

...

$ ./list_buckets.py

projectx-bucket1-2017-01-01-1483305884

---

        file1.txt

        file2.txt

        file3.txt

        file4.txt
```

Success! We've created a bucket and uploaded some files into it. Now let's go in the opposite direction, deleting objects and then finally, deleting the bucket.

DELETE BUCKET CONTENTS

For our delete script, we'll start the same as our create script: importing the needed libraries, creating an S3 resource, and taking bucket names as arguments.

To keep things simple, we'll delete all the objects in each bucket passed in as an argument. We'll wrap the call to the delete() function in a try: block to make sure we catch any errors. Our script looks like this:

```
#!/usr/bin/env python

import sys

import boto3
```

```
s3 = boto3.resource('s3')

for bucket_name in sys.argv[1:]:

    bucket = s3.Bucket(bucket_name)

    for key in bucket.objects.all():

        try:

            response = key.delete()

            print response

        except Exception as error:

            print error
```

If we save this as *./delete_contents.py* and run the script on our example bucket, output should look like this:

```
$ ./delete_contents.py projectx-bucket1-2017-01-01-1483305884

{'ResponseMetadata': {'HTTPStatusCode': 204, 'RetryAttempts': 0, 'HostId':
'+A4vDhUEyZgYUGSDHELJHWPt5xmZE2WNI/eqJVjYEsB6wCLU/i6a65sUSPK6x8PcoJXN/2oBmlQ=',
'RequestId': 'A097DD4C0413AF12', 'HTTPHeaders': {'x-amz-id-2':
'+A4vDhUEyZgYUGSDHELJHWPt5xmZE2WNI/eqJVjYEsB6wCLU/i6a65sUSPK6x8PcoJXN/2oBmlQ=',
'date': 'Sun, 01 Jan 2017 22:09:05 GMT', 'x-amz-request-id': 'A097DD4C0413AF12',
'server': 'AmazonS3'}}}

...
```

Now if we run the *list_buckets.py* script again, we'll see that our bucket is indeed empty.

```
$ ./list_buckets.py

projectx-bucket1-2017-01-01-1483305884

---
```

DELETE A BUCKET

Our delete bucket script looks a lot like our delete object script. The same libraries are imported and the arguments are taken to be bucket names. We use the S3 resource to attach to a bucket with the specific name and then in our try: block, we call the delete() function on that bucket, catching the response. If the delete worked, we print the response. If not, we print the error message. Here's the script:

```
#!/usr/bin/env python

import sys
import boto3

s3 = boto3.resource('s3')

for bucket_name in sys.argv[1:]:

    bucket = s3.Bucket(bucket_name)

try:

        response = bucket.delete()

print response

except Exception as error:

print error
```

One important thing to note when attempting to delete a bucket is that the bucket must be empty first. If there are still objects in a bucket when you try to delete it, an error will be reported and the bucket will not be deleted.

Running our *delete_buckets.py* script on our target bucket produces the following output:

```
$ ./delete_buckets.py projectx-bucket1-2017-01-01-1483305884

{'ResponseMetadata': {'HTTPStatusCode': 204, 'RetryAttempts': 0, 'HostId':
'LrM1/W6tlGPiB1FQyDQ4lKWndzXkK1joo2PZUtFPWqtz7CjEIzFttaANM6lLK5Zbk4+6O2qDwqc=',
'RequestId': 'DEBF57021D1AD121', 'HTTPHeaders': {'x-amz-id-2':
'LrM1/W6tlGPiB1FQyDQ4lKWndzXkK1joo2PZUtFPWqtz7CjEIzFttaANM6lLK5Zbk4+6O2qDwqc=',
'date': 'Sun, 01 Jan 2017 22:27:51 GMT', 'x-amz-request-id': 'DEBF57021D1AD121',
'server': 'AmazonS3'}}}
```

We can run *list_buckets.py* again to see that our bucket has indeed been deleted.

SCRIPTING RDS

The Relational Database Service (RDS) simplifies the management of a variety of database types including MySQL, Oracle, Microsoft SQL, and Amazon's own Aurora DB. Let's take a look at how we can examine, create, and delete RDS instances using Python and boto3.

In this high level example, we'll just be looking at the instances — the virtual machines — that host databases but not the databases themselves.

LIST DB INSTANCES

Let's start with RDS by getting a listing of the database instances that are currently running in our account.

Of course, we'll need to import the boto3 library and create a connection to RDS. Instead of using a resource, though, we'll create an RDS client. Clients are similar to resources but operate at a lower level of abstraction.

Using the client, we can call the *describe_db_instances()* function to list the database instances in our account. With a handle to each database instance, we can print details like the master username, the endpoint used to connect to the instance, the port that the instance is listening on, and the status of the instance. The script looks like this:

```
#!/usr/bin/env python
import boto3
rds = boto3.client('rds')
try:
# get all of the db instances
    dbs = rds.describe_db_instances()
for db in dbs['DBInstances']:
print ("%s@%s:%s %s") % (
            db['MasterUsername'],
            db['Endpoint']['Address'],
            db['Endpoint']['Port'],
            db['DBInstanceStatus'])
except Exception as error:
print error
```

If you run this script but haven't created any database instances, the output will be an empty response which is expected. If you see any errors, go back and check the script for anything that might be out of order.

CREATE A DB INSTANCE

Creating a database instance requires quite a bit of input. At the least, the following information is required:

- A name, or identifier, for the instance; this must be unique in each region for the user's account
- A username for the admin or root account

- A password for the admin account
- The class or type the instance will be created as
- The database engine that the instance will use
- The amount of storage the instance will allocate for databases

In our previous scripts, we used *sys.argv* to capture the input we needed from the command line. For this example, let's just prefill the inputs in the script to keep things as simple as possible.

Also in this case, we'll use the *db.t2.micro* as our instance class and *mariadb* as our database engine to make sure our database instances runs in the AWS free tier if applicable. For a complete listing of the database instance classes and the database engines that each class can run, review the RDS documentation for more information.

Proceeding with the script, we'll import the boto3 library and created an RDS client. With the client, we can call the *create_db_instance()* function and pass in the arguments needed to create the instance. We save the response and print it if all goes well. Because the create function is wrapped in a *try:* block, we can also catch and print an error message if anything takes a wrong turn. Here's the code:

```
#!/usr/bin/env python

import boto3

rds = boto3.client('rds')

try:

    response = rds.create_db_instance(

        DBInstanceIdentifier='dbserver',

        MasterUsername='dbadmin',

        MasterUserPassword='abcdefg123456789',

        DBInstanceClass='db.t2.micro',

        Engine='mariadb',

        AllocatedStorage=5)

print response

except Exception as error:

print error
```

We can save this as *create_db_instance.py* and run the script directly without having to pass any arguments. If all goes well, we should see a response similar to the one below (truncated for brevity).

```
$ ./create_db_instance.py

{u'DBInstance': {u'PubliclyAccessible': True, u'MasterUsername': 'dbadmin',
u'MonitoringInterval': 0, u'LicenseModel': 'general-public-license',
u'VpcSecurityGroups': [{u'Status': 'active', u'VpcSecurityGroupId': 'sg-2950f150'}],
....
```

Even though our script may have finished successfully, it takes some time for the database instance to be created up to 5 minutes or more. It's important to give the RDS infrastructure time to get our instance up and running before attempting any further actions, or those follow-on actions might fail.

So after waiting for a moment, we can run out list_db_instances.py script to see the details for the database instance we just created.

```
$ ./list_db_instances.py

dbadmin@dbserver.co2fg22sgwmb.us-west-2.rds.amazonaws.com:3306 available
```

In this case, we see our admin account name, our instance identifier along with the RDS endpoint assigned to it, and the port our database instance is listening on. Awesome! These all match up to the input we specified in the script.

Now that we know our database instance is up and running, we can go about connecting to it, creating user accounts, databases, and tables inside databases. Of course, all of those things can be done programmatically with Python but they are also beyond the scope of this tutorial. Take a look at The Hitchhiker's Guide to Python for more information on using databases with Python.

DELETE A DB INSTANCE

Once we no longer need a database instance, we can delete it. Of course, we'll import the boto3 library and the sys library as well this time; we'll need to get the name of the instance to be deleted as an argument. After creating an RDS client, we can wrap the call to delete_db_instance() in a try: block. We catch the response and print it. If there are any exceptions, we print the error for analysis. Here's the script:

```
#!/usr/bin/env python

import sys

import boto3

db = sys.argv[1]
```

```
rds = boto3.client('rds')

try:

    response = rds.delete_db_instance(

        DBInstanceIdentifier=db,

        SkipFinalSnapshot=True)

print response

except Exception as error:

print error
```

Before we try to delete anything, let's run list_db_instances.py to get the name of the database instances that are available. Then we can run delete_db_instance.py on one of the names. Like the creation step, deleting a database takes some time. But if we run list_db_instances.py right away, we'll see the status of the instance change to "deleting." That procedure looks like this:

```
$ ./list_db_instances.py

dbadmin@dbserver.co2fg22sgwmb.us-west-2.rds.amazonaws.com:3306 available

$ ./delete_db_instance.py dbserver

{u'DBInstance': {u'PubliclyAccessible': True, u'MasterUsername': 'dbadmin',
u'MonitoringInterval': 0, u'LicenseModel': 'general-public-license',
u'VpcSecurityGroups': [{u'Status': 'active', u'VpcSecurityGroupId': 'sg-2950f150'}],

…

$ ./list_db_instances.py

dbadmin@dbserver.co2fg22sgwmb.us-west-2.rds.amazonaws.com:3306 deleting
```

Once the instance has been completely deleted, running the list script again should return an empty list.

WRAPPING UP AND NEXT STEPS

In this tutorial we've covered a lot! If you followed from the beginning, you've configured an environment to run Python and installed the boto3 library. You created and configured an account to access AWS resources from the command line. You’ve also written and used Python scripts to list, create and delete objects in EC2, S3, and RDS. That’s quite the accomplishment.

Still, there’s more to learn. The examples used this this tutorial just scratch the surface of what can be done in AWS with Python. Also, the examples use just the bare minimum in regards to service configuration and security. I encourage you to explore the documentation for each service to see how these examples can be made more robust and more secure before applying them in your development and production workflows.

If you have feedback or questions on this tutorial, please leave them in the comments below. I also encourage you to share your successes as you explore and master automating AWS with Python.