

Implementing Unit Testing

 googlecoursera.qwiklabs.com/focuses/47746

Introduction

Imagine one of your IT coworkers just retired and left a folder of scripts for you to use. One of the scripts, called **emails.py**, matches users to an email address and lets us easily look them up! For the most part, the script works great — you enter in an employee's name and their email is printed to the screen. But, for some employees, the output doesn't look quite right. Your job is to add a test to reproduce the bug, make the necessary corrections, and verify that all the tests pass to make sure the script works! Best of luck!

What you'll do

In this lab, you will:

- Write a simple test to check for basic functionality
- Write a test to check for edge cases
- Correct code with a try/except statement

You'll have 90 minutes to complete this lab.

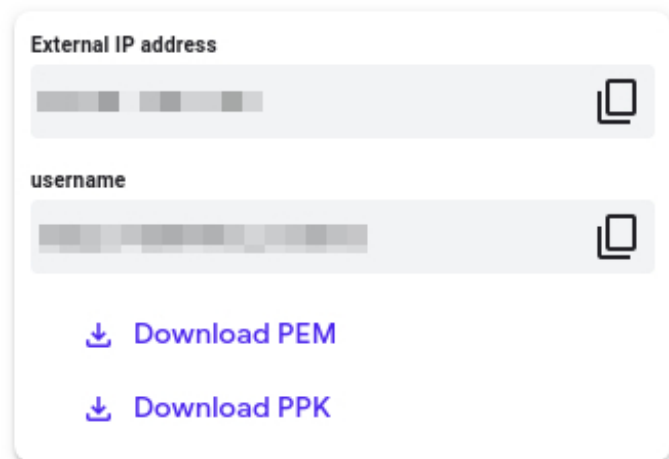
Start the lab

You'll need to start the lab before you can access the materials in the virtual machine OS. To do this, click the green “Start Lab” button at the top of the screen.

Note: For this lab you are going to access the **Linux VM** through your **local SSH Client**, and not use the **Google Console (Open GCP Console** button is not available for this lab).

After you click the “Start Lab” button, you will see all the SSH connection details on the left-hand side of your screen. You should have a screen that looks like this:





External IP address

username

Download PEM

Download PPK

Accessing the virtual machine

Please find one of the three relevant options below based on your device's operating system.

Note: Working with Qwiklabs may be similar to the work you'd perform as an **IT Support Specialist**; you'll be interfacing with a cutting-edge technology that requires multiple steps to access, and perhaps healthy doses of patience and persistence(!). You'll also be using **SSH** to enter the labs -- a critical skill in IT Support that you'll be able to practice through the labs.

Option 1: Windows Users: Connecting to your VM

In this section, you will use the PuTTY Secure Shell (SSH) client and your VM's External IP address to connect.

Download your PPK key file

You can download the VM's private key file in the PuTTY-compatible **PPK** format from the Qwiklabs Start Lab page. Click on **Download PPK**.

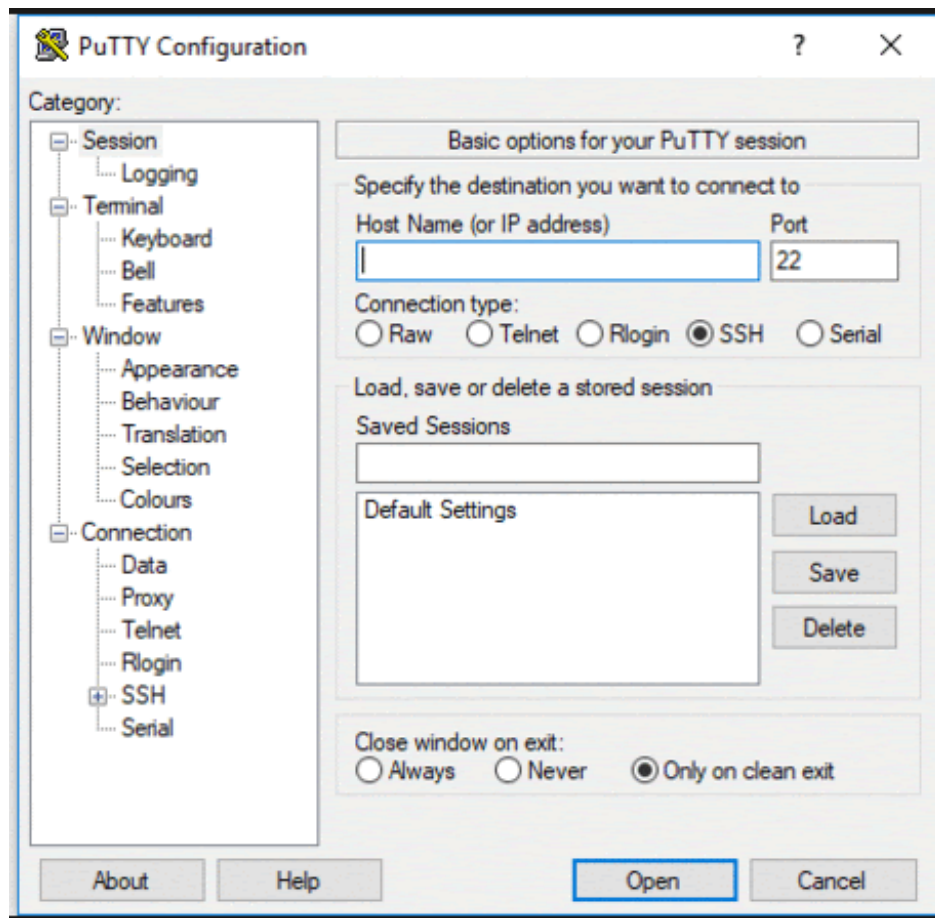
Connect to your VM using SSH and PuTTY

1. You can download Putty from [here](#)
2. In the **Host Name (or IP address)** box, enter `username@external_ip_address`.

Download PEM

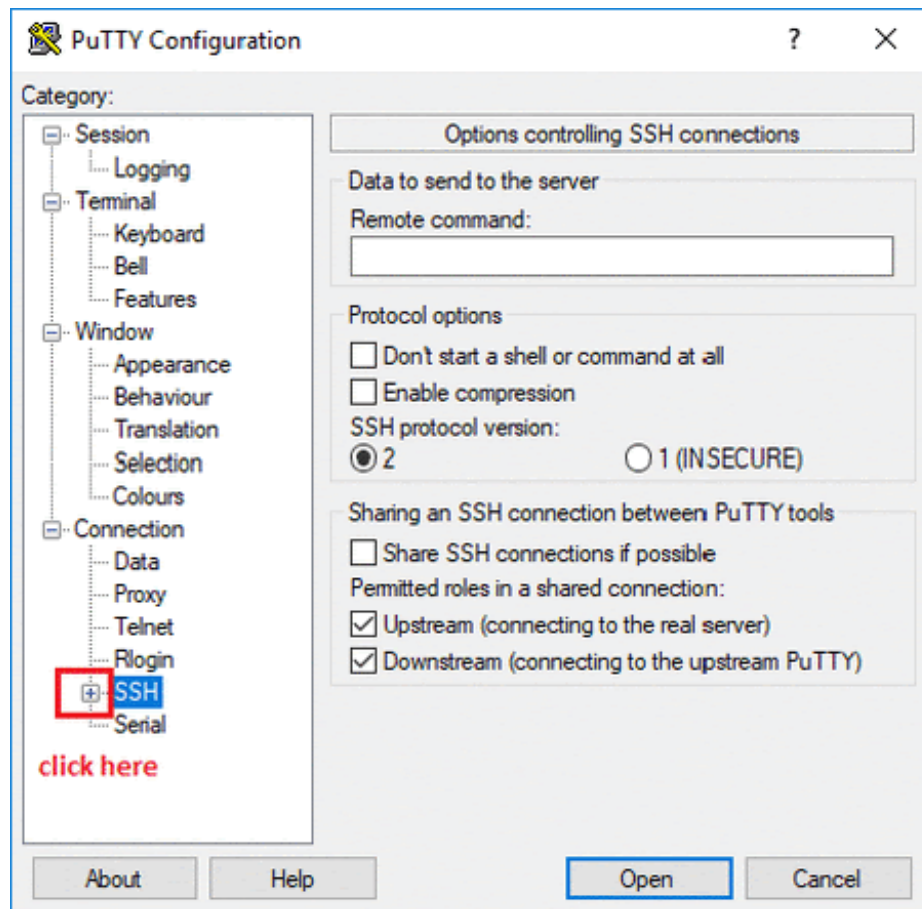
Download PPK

Note: Replace **username** and **external_ip_address** with values provided in the lab.



3. In the **Category** list, expand **SSH**.
4. Click **Auth** (don't expand it).
5. In the **Private key file for authentication** box, browse to the PPK file that you downloaded and double-click it.
6. Click on the **Open** button.

Note: PPK file is to be imported into PuTTY tool using the Browse option available in it. It should not be opened directly but only to be used in PuTTY.



7. Click **Yes** when prompted to allow a first connection to this remote SSH server. Because you are using a key pair for authentication, you will not be prompted for a password.

Common issues

If PuTTY fails to connect to your Linux VM, verify that:

- You entered `<username>@<external ip address>` in PuTTY.
- You downloaded the fresh new PPK file for this lab from Qwiklabs.
- You are using the downloaded PPK file in PuTTY.

Option 2: OSX and Linux users: Connecting to your VM via SSH

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.

Connect to the VM using the local Terminal application

A **terminal** is a program which provides a **text-based interface for typing commands**. Here you will use your terminal as an SSH client to connect with lab provided Linux VM.



1. Open the Terminal application.
 - To open the terminal in Linux use the shortcut key **Ctrl+Alt+t**.
 - To open terminal in **Mac** (OSX) enter **cmd + space** and search for **terminal**.
2. Enter the following commands.

Note: Substitute the **path/filename for the PEM** file you downloaded, **username** and **External IP Address**.

You will most likely find the PEM file in **Downloads**. If you have not changed the download settings of your system, then the path of the PEM key will be **~/Downloads/qwikLABS-XXXXX.pem**

```
chmod 600 ~/Downloads/qwikLABS-XXXXX.pem
```



```
ssh -i ~/Downloads/qwikLABS-XXXXX.pem username@External Ip Address
```



```
gcpstagingedit1370_student@35.239.106.192:~$ ssh -i ~/Downloads/qwikLABS-L923-42090.pem gcpstagingedit1370_student@35.239.106.192
The authenticity of host '35.239.106.192 (35.239.106.192)' can't be established.
ECDSA key fingerprint is SHA256:vrz8b4aYUtruFh0A6wZn60zy1oqqPEfh931o1vx1Tm8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.239.106.192' (ECDSA) to the list of known hosts.
Linux linux-instance 4.9.0-9-amd64 #1 SMP Debian 4.9.168-1+deb9u2 (2019-05-13) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
gcpstagingedit1370_student@linux-instance:~$
```

Option 3: Chrome OS users: Connecting to your VM via SSH

Note: Make sure you are not in **Incognito/Private mode** while launching the application.

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.

Connect to your VM

1. Add Secure Shell from [here](#) to your Chrome browser.
2. Open the Secure Shell app and click on **[New Connection]**.

A screenshot of the 'New Connection' dialog box in a Secure Shell application. The dialog has a title bar '[New Connection]'. Below it is a large text field labeled 'username@hostname or free form text'. Underneath are three input fields: 'username', 'hostname', and 'port'. Below these is a field for 'SSH relay server options'. Further down is an 'Identity:' dropdown menu currently set to '[default]', followed by an 'Import...' button. Below that is an 'SSH Arguments:' field with the text 'extra command line arguments'. Then a 'Current profile:' field set to 'default'. At the bottom is a 'Mount Path:' field with the text 'the default path is the user's home directory'. At the very bottom are buttons for '[DEL] Delete', 'Options', 'SFTP Mount', 'SFTP', '[ENTER] Connect', and 'Connect'.

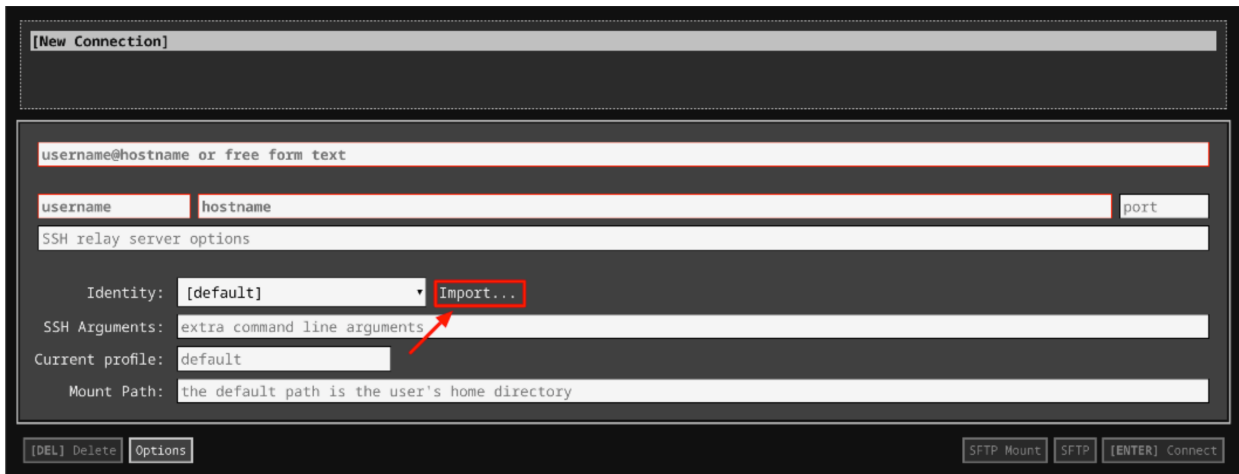
3. In the **username** section, enter the username given in the Connection Details Panel of the lab. And for the **hostname** section, enter the external IP of your VM instance that is mentioned in the Connection Details Panel of the lab.

A screenshot of the 'New Connection' dialog box, similar to the one above. In this version, the 'username' and 'hostname' input fields are highlighted with red rectangular boxes. A red arrow points from the bottom left towards the 'Import...' button.

4. In the **Identity** section, import the downloaded PEM key by clicking on the **Import...** button beside the field. Choose your PEM key and click on the **OPEN** button.

Note: If the key is still not available after importing it, refresh the application, and select it from the **Identity** drop-down menu.

5. Once your key is uploaded, click on the **[ENTER] Connect** button below.



6. For any prompts, type **yes** to continue.
7. You have now successfully connected to your Linux VM.

You're now ready to continue with the lab!

Prerequisites

First, you need to find the **.csv** file called **user_emails.csv**, which contains user names and their respective email addresses within the **data** directory. Navigate to this directory using the following command:

```
cd ~/data
```



List the files using the following command:

```
ls
```



You should now see a file named **user_emails.csv**. To view the contents of the **user_emails.csv** file, enter the following command:

```
cat user_emails.csv
```



Your IT coworker has also left a script named **emails.py** within the **scripts** directory.

Use the following command to navigate to the **scripts** directory:

```
cd ~/scripts
```



Now list the contents within the **scripts** directory using the following command:

```
ls
```



Here, you will find the script named **emails.py**. This script aims to match users to their respective email addresses.

You can view the file using the following command:

```
cat emails.py
```



This script consists of two functions: `populate_dictionary(filename)` and `find_email(argv)`. The function *populate_dictionary(filename)* reads the `user_emails.csv` file and populates a dictionary with name/value pairs. The other function, *find_emails(argv)*, searches the dictionary created in the previous function for the user name passed to the function as a parameter. It then returns the associated email address. This script accepts employee's first name and last name as command-line arguments and outputs their email address.

The script accepts arguments through the command line. These arguments are stored in a list named **sys.argv**. The first element of this list, i.e. `argv[0]`, is always the name of the file being executed. So the parameters, i.e., first name and last name, are then stored in `argv[1]` and `argv[2]` respectively.

Let's test the script now.

Since you know the contents of the **user_emails.csv** file, choose any name to be passed as a parameter, or you can use the following name:

```
python3 emails.py Blossom Gill
```



This will give you the email address associated with the Full Name passed as parameters. In this case, the name is Blossom Gill and the email ID associated with this name is `blossom@abc.edu`.

```
student-00-c24eec123753@linux-instance:~/scripts$ python3 emails.py Blossom Gill  
blossom@abc.edu
```

That was simple and straightforward. But this script has few bugs. In the next part of this lab, we will design some test cases and correct the bugs in the script.

Introduction to test cases

Writing a test encourages you to think through the script's design and goals before writing the code. This keeps you focused and lets you create better designs. If you learn how to easily test your scripts, you'll be able to create code that's better defined and cohesive.

In this lab, we will write tests and correct bugs within the existing script.

In this section, we will write a basic test case and see how it works. A test case is an individual unit of testing that checks for a specific response to a particular set of inputs.

Use the following command to create a new file (in scripts directory) to write our test cases:

```
nano ~/scripts/emails_test.py
```



The file should now open in edit mode. This script's primary objective is to write test cases that correct bugs in the existing emails.py script. We will use the unittest package for this.

Add the following shebang line and import the necessary packages:

```
#!/usr/bin/env python3
import unittest
```



The package **unittest** supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. This module also provides classes that make it simple to support these qualities for a set of tests.

The following import statement allows a Python file to access the script from another Python file. In this case, we will import the function **find_email**, which is defined in the script **emails.py**.

```
from emails import find_email
```



Now let's create a class:

```
class EmailsTest(unittest.TestCase):
```



Classes are a way to bundle data and functionality together. Creating a new class creates a new type of object, which further allows new instances of that type to be made.

Another important aspect of the unittest module is the test runner. A test runner is a component that orchestrates the execution of tests and provides the outcome to the user.

A test case is created by subclassing **unittest.TestCase**. Let's write our first basic test case, **test_basic**.

```
def test_basic(self):
    testcase = [None, "Bree", "Campbell"]
    expected = "breee@abc.edu"
    self.assertEqual(find_email(testcase), expected)
if __name__ == '__main__':
    unittest.main()
```

Here, variable **test case** contains the parameters to be passed to the script emails.py. As we mentioned, the script file is the first element of input parameters through command line using argv. Since we already imported the function find_email from emails.py earlier, we will pass None in place of the script file and call it later in the script. Adding to None, we will pass a first name and last name as parameters.

The variable stores the expected value to be returned by emails.py. The method assertEquals passes the test case to the function find_email, which we imported earlier from emails.py, and checks whether it generates the expected output.

Save the file by clicking Ctrl-o, Enter key, and Ctrl-x.

We will run this file through the command line here. To do this, we will give the file permissions for execution.

```
chmod +x emails_test.py
```

Now, let's run our first test case using the following command:

```
./emails_test.py
```

The output shows the number of tests run and its associated output.

```
student-00-9679d8b83ee1@linux-instance:~/scripts$ python3 emails_test.py
/home/student-00-9679d8b83ee1/scripts/emails.py:9: ResourceWarning: unclosed file <_io.TextIOWrapper name='/home/student-00-9679d8b83ee1/data/user_emails.csv' mode='r' encoding='UTF-8'>
  user_data = list(csv.DictReader(open(csv_file_location)))
. I
-----
Ran 1 test in 0.001s
OK
student-00-9679d8b83ee1@linux-instance:~/scripts$
```

The test case passed. This was a basic test case to show how test cases with Python work. In the next section, we will write a few more test cases covering other possibilities.

Test Case 1: Missing parameters

Imagine a scenario where the user doesn't give either their first name or last name. What do you think the output would be in this case?

Lets try this out. Choose any first or last name of your choice or use the following name to be passed to **emails.py** as a parameter:

python3 emails.py Kirk

```
student-00-9679d8b83ee1@linux-instance:~/scripts$ python3 emails.py Kirk
Traceback (most recent call last):
  File "emails.py", line 15, in <module>
    print(find_email(sys.argv))
  File "emails.py", line 12, in find_email
    found_email = [data['Email Address'] for data in user_data if data['Full Name'] == full_name][0]
IndexError: list index out of range
student-00-9679d8b83ee1@linux-instance:~/scripts$
```

This now gives us an error. The script doesn't take just one parameter as input and so it produces an error.

Let's now write a test case to handle this type of error. This test case should pass just the first name to the script.

nano emails_test.py

Add the test case **test_one_name** just after the first test case.

Pro tip: Note down the name of the test cases. Knowing the names will be helpful in running individual tests.

```
def test_one_name(self):
    testcase = [None, "John"]
    expected = "Missing parameters"
    self.assertEqual(find_email(testcase), expected)
```

The file **emails_test.py** should now look like this:

```
#!/usr/bin/env python3

import unittest
from emails import find_email

class TestFile(unittest.TestCase):
    def test_basic(self):
        testcase = [None, "Bree", "Campbell"]
        expected = "breee@abc.edu"
        self.assertEqual(find_email(testcase), expected)

    def test_one_name(self):
        testcase = [None, "John"]
        expected = "Missing parameters"
        self.assertEqual(find_email(testcase), expected)

if __name__ == '__main__':
    unittest.main()
```

Save the file by clicking Ctrl-o, Enter key, and Ctrl-x.

Now run the second test using the following command:

```
./emails_test.py
```

Another way to run a particular function within the script is to specify the class name and the function name you want to run. This helps us run individual tests without having to run all the test cases in the test script again.

This now returns the following output:

```
student-02-9c7ef4fa6991@linux-instance:~/scripts$ ./emails_test.py
.E
=====
ERROR: test_one_name (__main__.TestFile)
-----
Traceback (most recent call last):
  File "./emails_test.py", line 15, in test_one_name
    self.assertEqual(find_email(testcase), expected)
  File "/home/student-02-9c7ef4fa6991/scripts/emails.py", line 19, in find_email
    fullname = str(argv[1] + " " + argv[2])
IndexError: list index out of range

-----
Ran 2 tests in 0.001s
FAILED (errors=1)
```

The output shows the function that caused the error and the description related to the error. It returned `IndexError`, which is raised while attempting to access an index that's outside the bounds of a list. This error occurs because the script **emails.py** takes in two parameters, the first and last name. We need to handle this type of incomplete inputs within the script. We need to decide what the correct output should be. Let's say, in this case, your script should output "Missing parameter".

Let's now fix the code. The last test case showed that the script fails if only one parameter is passed. We would now handle these types of incomplete inputs given to the script file **emails.py**.

There are two ways to solve this issue:

- Use a try/except clause to handle `IndexError`.
- Check the length of input parameters before traversing the `user_emails.csv` file for the email address.

You can use either of the above methods, but remember that test cases should pass and the script should return "Missing parameters" in this case.

We will use the try/except clause here to solve this issue. Try/except blocks are used for exceptions and error handling. Since exceptions are detected during execution of a script/program, error handling in Python is done using exceptions that are caught in try blocks and handled in except blocks.

Let's dive into how try/except blocks work:

- First, we execute the try clause.
- If no exception occurs, the except clause is ignored.
- If an exception occurs during the execution of the try clause, the rest of the try clause is then skipped.
- It then attempts to match the type with the exception named after the **except** keyword. If this matches, the except clause is executed. If it doesn't, the control is passed on to outer try statements. If no handler is found, it's an unhandled exception and the execution stops with an error message.

A try statement may have more than one except clause to specify handlers for different exceptions. In our case, the exception error we need to handle is **IndexError**.

Let's move forward by adding a try/except clause to the script **emails.py**.

nano emails.py



We will add the complete code block within the function **find_email(argv)**, which is within the try block, and add an IndexError exception within the except block. This means that the execution will start normally with any number of parameters given to the script. If the function **find_email(argv)** receives the required number of parameters, it will return the email address. And if the function doesn't receive the required number of parameters, it will throw an IndexError exception and the except clause which handles IndexError exception would then execute.

Add the body of the function **find_emails(argv)** within the try block and add an except block:

```
def find_email(argv):
    """ Return an email address based on the username given."""
    # Create the username based on the command line input.
    try:
        fullname = str(argv[1] + " " + argv[2])
        # Preprocess the data
        email_dict = populate_dictionary('/home/<username>/data/user_emails.csv')
        # Find and print the email
        return email_dict.get(fullname.lower())
    except IndexError:
        return "Missing parameters"
```



The complete file emails.py should now look like this:

```
#!/usr/bin/env python3

import sys
import csv

def populate_dictionary(filename):
    """Populate a dictionary with name/email pairs for easy lookup."""
    email_dict = {}
    with open(filename) as csvfile:
        lines = csv.reader(csvfile, delimiter = ',')
        for row in lines:
            name = str(row[0].lower())
            email_dict[name] = row[1]
    return email_dict

def find_email(argv):
    """ Return an email address based on the username given."""
    # Create the username based on the command line input.
    try:
        fullname = str(argv[1] + " " + argv[2])
        # Preprocess the data
        email_dict = populate_dictionary('/home/{ { username } }/data/user_emails.csv')
        # Find and print the email
        return email_dict.get(fullname.lower())
    except IndexError:
        return "Missing parameters"

def main():
    print(find_email(sys.argv))

if __name__ == "__main__":
    main()
```

Save the file by clicking Ctrl-o, Enter key, and Ctrl-x.

Now run the test cases within the file email_test.py again:

```
./emails_test.py
```

You should now see that both the test cases ran successfully and an OK message appeared.

Click *Check my progress* to verify the objective. Test case 1: Missing parameters

Congrats! You've just handled a test case within the script.

Test Case 2: Random email address

Let's find some other edge cases. We'll search for an employee that doesn't exist. Can you expect the output the script would give? The expected output in such a case should be "No email address found". Let's see how the script reacts to this case by adding a test case in the file **emails_test.py** just after the second test case.

Open the file `emails_test.py`.

nano `emails_test.py`



Add the following test case after the previous test case:

```
def test_two_name(self):
    testcase = [None, "Roy", "Cooper"]
    expected = "No email address found"
    self.assertEqual(find_email(testcase), expected)
```



The file should now look like this:

```
#!/usr/bin/env python3
```

```
import unittest
from emails import find_email
```

```
class EmailsTest(unittest.TestCase):
    def test_basic(self):
        testcase = [None, "Bree", "Campbell"]
        expected = "breee@abc.edu"
        self.assertEqual(find_email(testcase), expected)

    def test_one_name(self):
        testcase = [None, "John"]
        expected = "Missing parameters"
        self.assertEqual(find_email(testcase), expected)

    def test_two_name(self):
        testcase = [None, "Roy", "Cooper"]
        expected = "No email address found"
        self.assertEqual(find_email(testcase), expected)

if __name__ == '__main__':
    unittest.main()
```



Save the file by clicking Ctrl-o, Enter key, and Ctrl-x.

Run the test script using:

```
./emails_test.py
```



The test case failed! This means the script doesn't output the message "No email address found" if we search for an employee that doesn't exist.

Let's edit the script **emails.py** to return a message saying "No email address found" where users searched for don't exist.

Can you guess the statement where the function `find_email(argv)` actually fetches the email address of the user? The method **`email_dict.get(full)`**: does the job. This method fetches the email address from the list if found, and if not, it returns `None`.

We need to add an if-else loop here, which will return the email address only if the method `email_dict.get(username)` returns a valid email address. If it doesn't, it will return the message "No email address found".

To do this, edit the script file using:

```
nano emails.py
```



Locate the statement **`return email_dict.get(fullname.lower())`**: within the script under the function `find_email(argv)` and replace it with the following block of code:

```
if email_dict.get(fullname.lower()):  
    return email_dict.get(fullname.lower())  
else:  
    return "No email address found"
```



The file should now look like this:

```
#!/usr/bin/env python3

import csv
import sys

def populate_dictionary(filename):
    """Populate a dictionary with name/email pairs for easy lookup."""
    email_dict = {}
    with open(filename) as csvfile:
        lines = csv.reader(csvfile, delimiter = ',')
        for row in lines:
            name = str(row[0].lower())
            email_dict[name] = row[1]
    return email_dict

def find_email(argv):
    """ Return an email address based on the username given."""
    # Create the username based on the command line input.
    try:
        fullname = str(argv[1] + " " + argv[2])
        # Preprocess the data
        email_dict = populate_dictionary('/home/{ { username } }/data/user_emails.csv')
        # If email exists, print it
        if email_dict.get(fullname.lower()):
            return email_dict.get(fullname.lower())
        else:
            return "No email address found"
    except IndexError:
        return "Missing parameters"

def main():
    print(find_email(sys.argv))

if __name__ == "__main__":
    main()
```

Save the file by clicking Ctrl-o, Enter key, and Ctrl-x.

Now, run the test case to check if the script still produces an error.

```
python3 emails_test.py
```

Since we've handled the IndexError exception, the test case should now pass.

```
student-02-9c7ef4fa6991@linux-instance:~/scripts$ python3 emails_test.py
...
-----
Ran 3 tests in 0.001s
OK
```

You can also run the script **emails.py** by passing some random names (that aren't present in `user_emails.csv`) and check the output.

```
python3 emails.py Roy Cooper
```



This should now give the following output:

```
student-00-9679d8b83ee1@linux-instance:~/scripts$ python3 emails.py Roy Cooper
No email address found
student-00-9679d8b83ee1@linux-instance:~/scripts$
```

Click *Check my progress* to verify the objective. Test case 2: Random email address

Congratulations!

Congrats! You've successfully added tests to reproduce bugs, made the necessary corrections, and verified all tests pass to make sure the script works! Great job!

End your lab

When you have completed your lab, click **End Lab**. Qwiklabs removes the resources you've used and cleans the account for you.

You will be given an opportunity to rate the lab experience. Select the applicable number of stars, type a comment, and then click **Submit**.

The number of stars indicates the following:

- 1 star = Very dissatisfied
- 2 stars = Dissatisfied
- 3 stars = Neutral
- 4 stars = Satisfied
- 5 stars = Very satisfied

You can close the dialog box if you don't want to provide feedback.

For feedback, suggestions, or corrections, please use the **Support** tab.

