**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.



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 assertEqual 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.

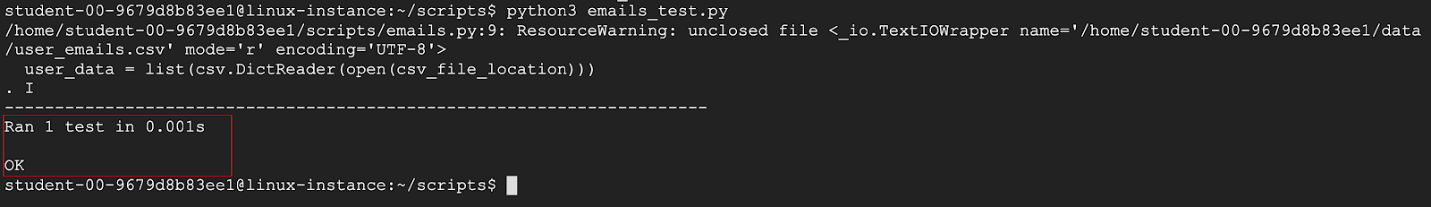
We will run this file through the command line here. To do this, we will give the file permissions for execution.

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.



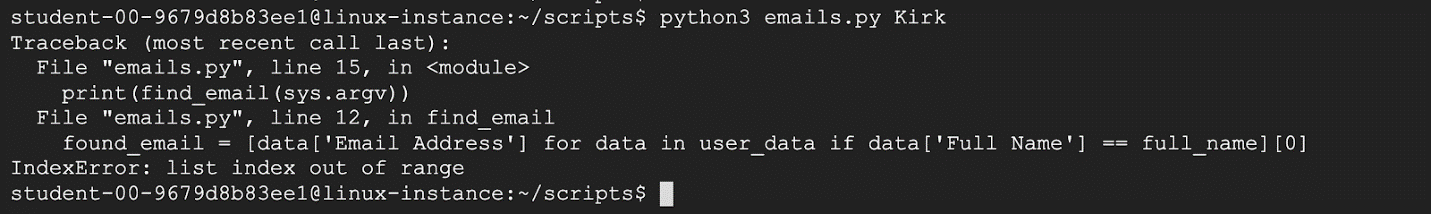
The test case passed. This was a basic test case to show how test cases with Python work. In the next section, we will write a few more test cases covering other possibilities.

**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



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()

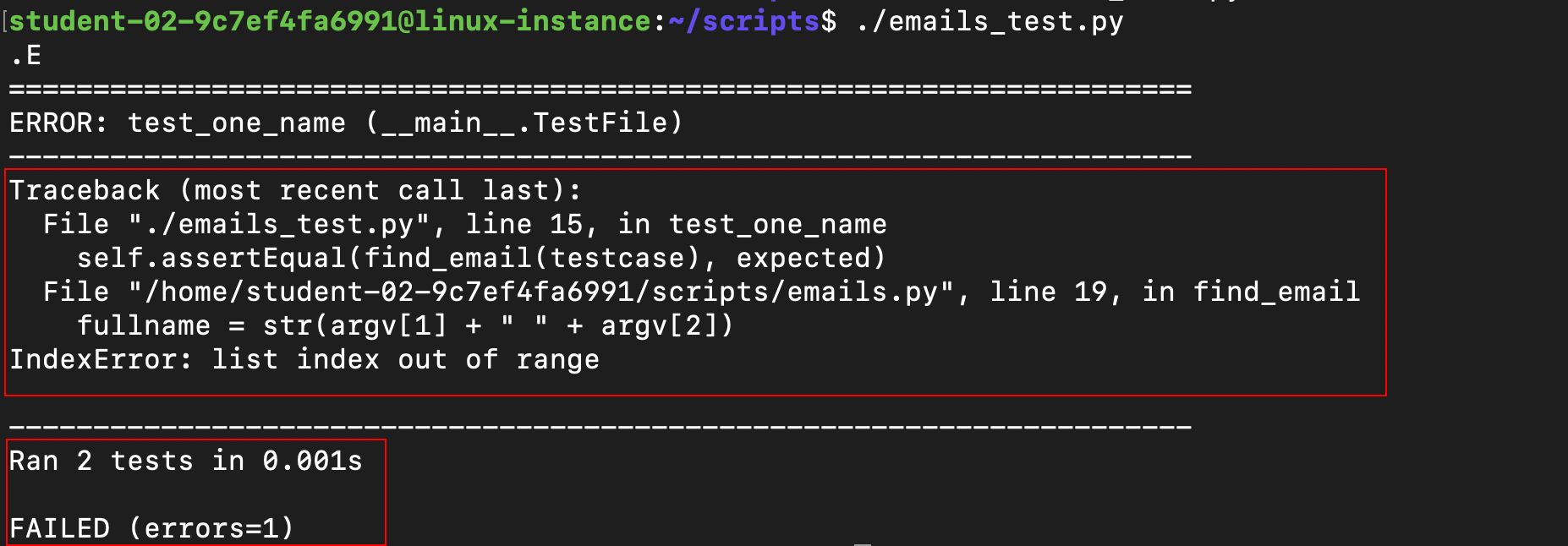
Save the file by clicking Ctrl-o, Enter key, and Ctrl-x.

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:



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.