“ *E-mail Slicer* ”

***Introduction:***

An easy-to-use tool, called an email slicer, can be utilized to extract both the username and the domain name from an email address. The Email Slicer tool takes an email address as input and performs slicing operations on it to retrieve the username and the domain. This tool can be particularly useful for swiftly identifying the domain name of an email address and sorting a collection of email addresses by their respective domain names.

The portion of the email address appearing before the "@" symbol is referred to as the username. This username serves as a unique identifier for the email account. On the other hand, the part of the email address that follows the "@" symbol is known as the domain name. The domain name specifies the mail server responsible for handling email operations for that account.

***Use Case of E-mail Slicer:***

Especially in industries like marketing, sales, and customer segmentation, the Email Slicer proves to be an invaluable tool for data analysis. Organizing email addresses based on their domain names can provide valuable insights into the characteristics and preferences of specific consumer or client groups. This tool assists in data analysis by enabling the classification and arrangement of email addresses according to their domain names, thus benefiting customer, market, and sales segmentation efforts.

Consider a collection of email addresses belonging to customers or clients – the Email Slicer can efficiently sort them by domain name. This process could yield significant information about the distinct characteristics and preferences of particular clienteles. Subsequently, this data can be leveraged to refine marketing messages, enhancing their effectiveness in resonating with the intended target market.

Top of Form

***Creating E-mail Slicer Using Python:***

***For a single e-mail id-***

*Code~*

email = input("Enter your email address: ")

username = email[:email.index("@")]

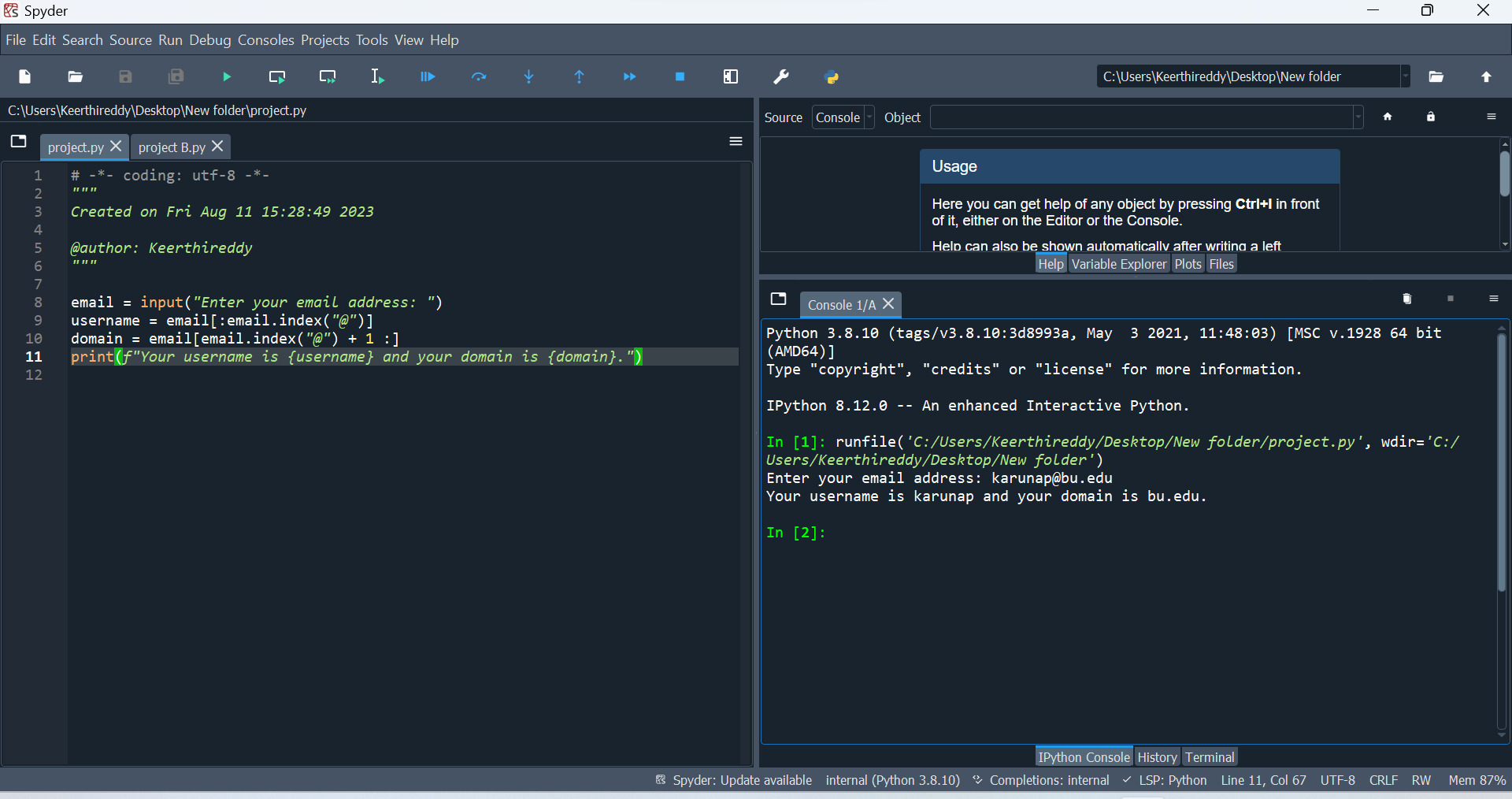
domain = email[email.index("@") + 1 :]

print(f"Your username is {username} and your domain is {domain}.")

*Given E-mail Id~*

karunap@bu.edu

*After Execution~*



***For a list of email id’s:***

*Code~*

import csv

class EmailProcessor:

def \_\_init\_\_(self, input\_file):

self.\_input\_file = input\_file

self.emails = [] # Public instance attribute

def \_read\_emails(self):

try:

with open(self.\_input\_file, 'r') as csv\_file:

reader = csv.reader(csv\_file)

next(reader) # Skip the header row if present

self.emails = [row[0] for row in reader]

except FileNotFoundError:

print("Input file not found.")

def slice\_email(self, email):

username, domain = email.split('@')

return username, domain

def process\_emails(self):

sliced\_emails\_list = []

for email in self.emails:

username, domain = self.slice\_email(email)

sliced\_emails\_list.append([username, domain])

return sliced\_emails\_list

def save\_to\_csv(self, data, output\_file):

with open(output\_file, 'w', newline='') as csv\_file:

writer = csv.writer(csv\_file)

writer.writerow(['Username', 'Domain'])

writer.writerows(data)

def process\_and\_save(self, output\_file):

try:

self.\_read\_emails() # Using the private method

sliced\_emails\_list = self.process\_emails()

sorted\_sliced\_emails = sorted(sliced\_emails\_list, key=lambda x: x[1]) # Sort by domain

self.save\_to\_csv(sorted\_sliced\_emails, output\_file)

except Exception as e:

print(f"An error occurred: {e}")

else:

print("Email processing and saving completed successfully.")

def \_\_repr\_\_(self):

return f"EmailProcessor(input\_file={self.\_input\_file})"

@classmethod

def example\_class\_method(cls):

print("This is an example class method.")

# Unit Tests

import unittest

class TestEmailProcessor(unittest.TestCase):

def test\_process\_and\_save\_with\_valid\_input(self):

self.create\_test\_input()

email\_processor = EmailProcessor(self.input\_file)

email\_processor.process\_and\_save(self.output\_file)

with open(self.output\_file, 'r') as csv\_file:

reader = csv.reader(csv\_file)

header = next(reader)

self.assertEqual(header, ['Username', 'Domain'])

actual\_sliced\_emails = [row for row in reader]

self.assertEqual(actual\_sliced\_emails, self.expected\_sliced\_emails)

print("Test for process\_and\_save with valid input passed.")

def test\_process\_and\_save\_with\_invalid\_input(self):

# Create an empty input file

open(self.input\_file, 'w').close()

email\_processor = EmailProcessor(self.input\_file)

email\_processor.process\_and\_save(self.output\_file)

with open(self.output\_file, 'r') as csv\_file:

reader = csv.reader(csv\_file)

header = next(reader, None)

self.assertIsNone(header)

print("Test for process\_and\_save with invalid input passed.")

def test\_slice\_email(self):

email\_processor = EmailProcessor(self.input\_file)

for email, expected\_sliced\_email in zip(self.emails, self.expected\_sliced\_emails):

sliced\_email = email\_processor.slice\_email(email)

self.assertEqual(sliced\_email, tuple(expected\_sliced\_email))

print("Test for slice\_email passed.")

def main():

input\_file = 'C:\\Users\\Keerthireddy\\Desktop\\email slicer.csv' # Replace with the actual path to your CSV file

email\_processor = EmailProcessor(input\_file) email\_processor.process\_and\_save('C:\\Users\\Keerthireddy\\Desktop\\sliced\_emails.csv')

print("Unique domains:")

unique\_domains\_set = set()

for \_, domain in email\_processor.process\_emails():

unique\_domains\_set.add(domain)

# Using an if conditional to check for a specific domain

specific\_domain = 'harvard.edu'

if specific\_domain in unique\_domains\_set:

print(f"The domain {specific\_domain} is present in the unique domains set.")

else:

print(f"The domain {specific\_domain} is not present in the unique domains set.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

*Explanation for the code~*

import csv

class EmailProcessor:

def \_\_init\_\_(self, input\_file):

self.\_input\_file = input\_file

self.emails = [] # Public instance attribute

def \_read\_emails(self):

try:

with open(self.\_input\_file, 'r') as csv\_file:

reader = csv.reader(csv\_file)

next(reader) # Skip the header row if present

self.emails = [row[0] for row in reader]

except FileNotFoundError:

print("Input file not found.")

**import csv**: This imports the built-in **csv** module, which provides functionality for reading and writing CSV (Comma-Separated Values) files.

**class EmailProcessor:**: This defines the **EmailProcessor** class.

**def \_\_init\_\_(self, input\_file):**: The constructor method (also called **\_\_init\_\_**) is called when an object of the class is created. It takes an **input\_file** parameter, which should be the path to a CSV file containing email addresses. It initializes two attributes: **\_input\_file** (with a leading underscore indicating it's a private attribute) and **emails** (a list to store the email addresses).

**\_read\_emails(self):**: This is a private method that attempts to read the email addresses from the CSV file specified in the **\_input\_file** attribute. It uses a **try** block to handle the case where the file is not found. It uses the **csv.reader** to iterate through the rows of the CSV file and populate the **emails** list with the first element of each row (assuming it's an email address).

**except FileNotFoundError:**: This block is executed if the specified CSV file is not found. It prints an error message indicating that the input file was not found.

**def slice\_email(self, email):**: This method takes an **email** parameter and splits it into two parts: the username and the domain. It returns these two parts as a tuple.

**def process\_emails(self):**: This method processes the list of email addresses stored in the **emails** attribute. It iterates through the list and uses the **slice\_email** method to split each email address into username and domain parts. It appends these parts to a list and returns a list of lists (sliced email addresses).

**def save\_to\_csv(self, data, output\_file):**: This method takes **data** (a list of lists) and an **output\_file** path as parameters. It uses the **csv.writer** to write the data to a CSV file specified by **output\_file**.

**def process\_and\_save(self, output\_file):**: This method processes the emails using the **\_read\_emails** and **process\_emails** methods, then saves the processed emails to a CSV file using the **save\_to\_csv** method. If an error occurs during the process, it catches the exception and prints an error message.

**def \_\_repr\_\_(self):**: This method provides a string representation of the **EmailProcessor** object. It returns a formatted string containing the input file path.

def slice\_email(self, email):

username, domain = email.split('@')

return username, domain

def process\_emails(self):

sliced\_emails\_list = []

for email in self.emails:

username, domain = self.slice\_email(email)

sliced\_emails\_list.append([username, domain])

return sliced\_emails\_list

def save\_to\_csv(self, data, output\_file):

with open(output\_file, 'w', newline='') as csv\_file:

writer = csv.writer(csv\_file)

writer.writerow(['Username', 'Domain'])

writer.writerows(data)

def slice\_email(self, email):

This method takes an email as input, which is assumed to be in the format "username@domain".

It uses the split('@') method to split the email into two parts: username and domain.

The split('@') method divides the string at the "@" symbol and returns a list containing the two parts.

This method returns a tuple containing username and domain.

def process\_emails(self):

This method processes the list of email addresses stored in the emails attribute of the class instance.

It initializes an empty list sliced\_emails\_list to store the sliced email addresses.

It iterates through each email in the emails list using a for loop.

For each email, it calls the slice\_email method to split the email into username and domain parts.

It appends a new list containing [username, domain] to the sliced\_emails\_list.

After processing all emails, it returns the sliced\_emails\_list containing the sliced email addresses as lists.

def save\_to\_csv(self, data, output\_file):

This method takes data (a list of lists containing sliced email addresses) and an output\_file path as parameters.

It uses a with statement to open the output\_file in write mode. The newline='' argument is used to ensure consistent line endings in the CSV file across different platforms.

It creates a csv.writer object named writer associated with the opened file.

It writes the header row to the CSV file using writer.writerow(['Username', 'Domain']).

It then writes the data from the data list (which contains sliced email addresses) to the CSV file using writer.writerows(data).

def process\_and\_save(self, output\_file):

try:

self.\_read\_emails() # Using the private method

sliced\_emails\_list = self.process\_emails()

sorted\_sliced\_emails = sorted(sliced\_emails\_list, key=lambda x: x[1]) # Sort by domain

self.save\_to\_csv(sorted\_sliced\_emails, output\_file)

except Exception as e:

print(f"An error occurred: {e}")

else:

print("Email processing and saving completed successfully.")

def \_\_repr\_\_(self):

return f"EmailProcessor(input\_file={self.\_input\_file})"

@classmethod

def example\_class\_method(cls):

print("This is an example class method.")

def process\_and\_save(self, output\_file):

This method processes the emails using the \_read\_emails and process\_emails methods, then sorts the sliced emails by domain and saves the sorted data to a CSV file using the save\_to\_csv method.

It uses a try block to attempt the process.

Inside the try block:

It calls the private method \_read\_emails() to read the email addresses from the input CSV file and populate the emails attribute.

It calls the process\_emails method to create a list of lists containing the sliced email addresses.

It uses the sorted() function to sort the sliced\_emails\_list based on the domain (index 1 of each sublist) using a lambda function.

Finally, it calls the save\_to\_csv method to save the sorted sliced emails to the specified output\_file.

If an exception occurs during the process (any exception derived from the Exception base class), it will catch the exception and print an error message along with the exception details.

If the process completes successfully (no exceptions are raised), the else block will be executed, printing a success message.

def \_\_repr\_\_(self):

This is a special method that provides a string representation of the EmailProcessor object. It returns a string in the format "EmailProcessor(input\_file=<input\_file\_path>)", where <input\_file\_path> is the actual input file path.

@classmethod

This is a decorator that defines a class method, which means the method is bound to the class itself rather than an instance of the class.

def example\_class\_method(cls):

This class method is defined using the @classmethod decorator.

It takes one parameter, conventionally named cls, which refers to the class itself.

Inside the method, it simply prints the message "This is an example class method.".

# Unit Tests

import unittest

class TestEmailProcessor(unittest.TestCase):

def test\_process\_and\_save\_with\_valid\_input(self):

self.create\_test\_input()

email\_processor = EmailProcessor(self.input\_file)

email\_processor.process\_and\_save(self.output\_file)

with open(self.output\_file, 'r') as csv\_file:

reader = csv.reader(csv\_file)

header = next(reader)

self.assertEqual(header, ['Username', 'Domain'])

actual\_sliced\_emails = [row for row in reader]

self.assertEqual(actual\_sliced\_emails, self.expected\_sliced\_emails)

print("Test for process\_and\_save with valid input passed.")

This section of code defines a unit test class **TestEmailProcessor** using the **unittest.TestCase** class. It includes a test method **test\_process\_and\_save\_with\_valid\_input** that checks the behavior of the **process\_and\_save** method of the **EmailProcessor** class when provided with valid input data.

**import unittest**: This imports the **unittest** module, which provides tools for constructing and running test cases.

**class TestEmailProcessor(unittest.TestCase):**: This defines the test class **TestEmailProcessor** which inherits from **unittest.TestCase**.

**def test\_process\_and\_save\_with\_valid\_input(self):**: This defines a test method named **test\_process\_and\_save\_with\_valid\_input**.

Inside this test method:

**self.create\_test\_input()**: This line calls a method named **create\_test\_input()**. This method is not shown in the provided code snippet, but it's likely defined elsewhere in the code. It's responsible for creating a temporary test input CSV file for the test.

**email\_processor = EmailProcessor(self.input\_file)**: This line creates an instance of the **EmailProcessor** class using the **input\_file** specified. The **input\_file** is the path to the temporary test input CSV file.

**email\_processor.process\_and\_save(self.output\_file)**: This line calls the **process\_and\_save** method of the **EmailProcessor** instance. It processes the email addresses from the temporary input CSV file, sorts them by domain, and saves the results to a temporary output CSV file.

The next few lines open the temporary output CSV file, read its content, and compare it with the expected sliced email addresses. If they match, the test case passes; if not, it fails.

**self.assertEqual(header, ['Username', 'Domain'])**: This asserts that the header of the output CSV file matches the expected header **['Username', 'Domain']**.

**self.assertEqual(actual\_sliced\_emails, self.expected\_sliced\_emails)**: This asserts that the content of the output CSV file (the sliced email addresses) matches the expected sliced email addresses defined earlier in the test class.

**print("Test for process\_and\_save with valid input passed.")**: If the test passes, this line prints a success message to indicate that the test has passed.

def test\_process\_and\_save\_with\_invalid\_input(self):

# Create an empty input file

open(self.input\_file, 'w').close()

email\_processor = EmailProcessor(self.input\_file)

email\_processor.process\_and\_save(self.output\_file)

with open(self.output\_file, 'r') as csv\_file:

reader = csv.reader(csv\_file)

header = next(reader, None)

self.assertIsNone(header)

print("Test for process\_and\_save with invalid input passed.")

def test\_slice\_email(self):

email\_processor = EmailProcessor(self.input\_file)

for email, expected\_sliced\_email in zip(self.emails, self.expected\_sliced\_emails):

sliced\_email = email\_processor.slice\_email(email)

self.assertEqual(sliced\_email, tuple(expected\_sliced\_email))

print("Test for slice\_email passed.")

In this section, two more test methods are defined in the **TestEmailProcessor** class. def test\_process\_and\_save\_with\_invalid\_input(self):

This test method is designed to test the behavior of the process\_and\_save method when provided with invalid input data.

Inside this test method:

open(self.input\_file, 'w').close(): This line creates an empty input file. It uses the open function with the mode 'w' to open the input file in write mode and then immediately closes it using the close method. This effectively creates an empty file.

email\_processor = EmailProcessor(self.input\_file): This line creates an instance of the EmailProcessor class using the empty input file.

email\_processor.process\_and\_save(self.output\_file): This line calls the process\_and\_save method of the EmailProcessor instance. Since the input file is empty, there are no email addresses to process, and the output file should also be empty.

The next lines open the temporary output CSV file, read its content, and check if the header is None (indicating an empty file). If the header is None, it means that no rows were written to the output CSV file, which is expected given the empty input file.

self.assertIsNone(header): This asserts that the header of the output CSV file is None, indicating that the output file is empty.

print("Test for process\_and\_save with invalid input passed."): If the test passes, this line prints a success message to indicate that the test has passed.

def test\_slice\_email(self):

This test method is designed to test the behavior of the slice\_email method of the EmailProcessor class.

Inside this test method:

email\_processor = EmailProcessor(self.input\_file): This line creates an instance of the EmailProcessor class using the input file.

A loop is used to iterate over the test email addresses and their expected sliced forms.

sliced\_email = email\_processor.slice\_email(email): This line calls the slice\_email method of the EmailProcessor instance with a test email address. It assigns the result to the variable sliced\_email.

self.assertEqual(sliced\_email, tuple(expected\_sliced\_email)): This asserts that the sliced email returned by the slice\_email method matches the expected sliced email, which is a tuple.

print("Test for slice\_email passed."): If the test passes, this line prints a success message to indicate that the test has passed.

def main():

input\_file = 'C:\\Users\\Keerthireddy\\Desktop\\email slicer.csv' # Replace with the actual path to your CSV file

email\_processor = EmailProcessor(input\_file)

email\_processor.process\_and\_save('C:\\Users\\Keerthireddy\\Desktop\\sliced\_emails.csv')

print("Unique domains:")

unique\_domains\_set = set()

for \_, domain in email\_processor.process\_emails():

unique\_domains\_set.add(domain)

# Using an if conditional to check for a specific domain

specific\_domain = 'harvard.edu'

if specific\_domain in unique\_domains\_set:

print(f"The domain {specific\_domain} is present in the unique domains set.")

else:

print(f"The domain {specific\_domain} is not present in the unique domains set.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

This **main()** function is the entry point of your program. It demonstrates how to use the **EmailProcessor** class to process and save email addresses from a CSV file.

**input\_file = 'C:\\Users\\Keerthireddy\\Desktop\\email slicer.csv'**: This line defines the path to your input CSV file. Make sure to replace it with the actual path to your CSV file.

**email\_processor = EmailProcessor(input\_file)**: This line creates an instance of the **EmailProcessor** class, passing the **input\_file** path as an argument. This instance will be used to process and save email addresses.

**email\_processor.process\_and\_save('C:\\Users\\Keerthireddy\\Desktop\\sliced\_emails.csv')**: This line calls the **process\_and\_save** method of the **EmailProcessor** instance. It processes the email addresses from the input CSV file, sorts them by domain, and saves the results to the specified output CSV file path (**'C:\\Users\\Keerthireddy\\Desktop\\sliced\_emails.csv'**).

**print("Unique domains:")**: This line prints a header indicating that the unique domain analysis is about to be displayed.

**unique\_domains\_set = set()**: This creates an empty set to store unique domains.

The following loop iterates over the processed email addresses using the **process\_emails** method of the **EmailProcessor** instance.

**for \_, domain in email\_processor.process\_emails():**: This loop extracts the domain from each processed email address and adds it to the **unique\_domains\_set**.

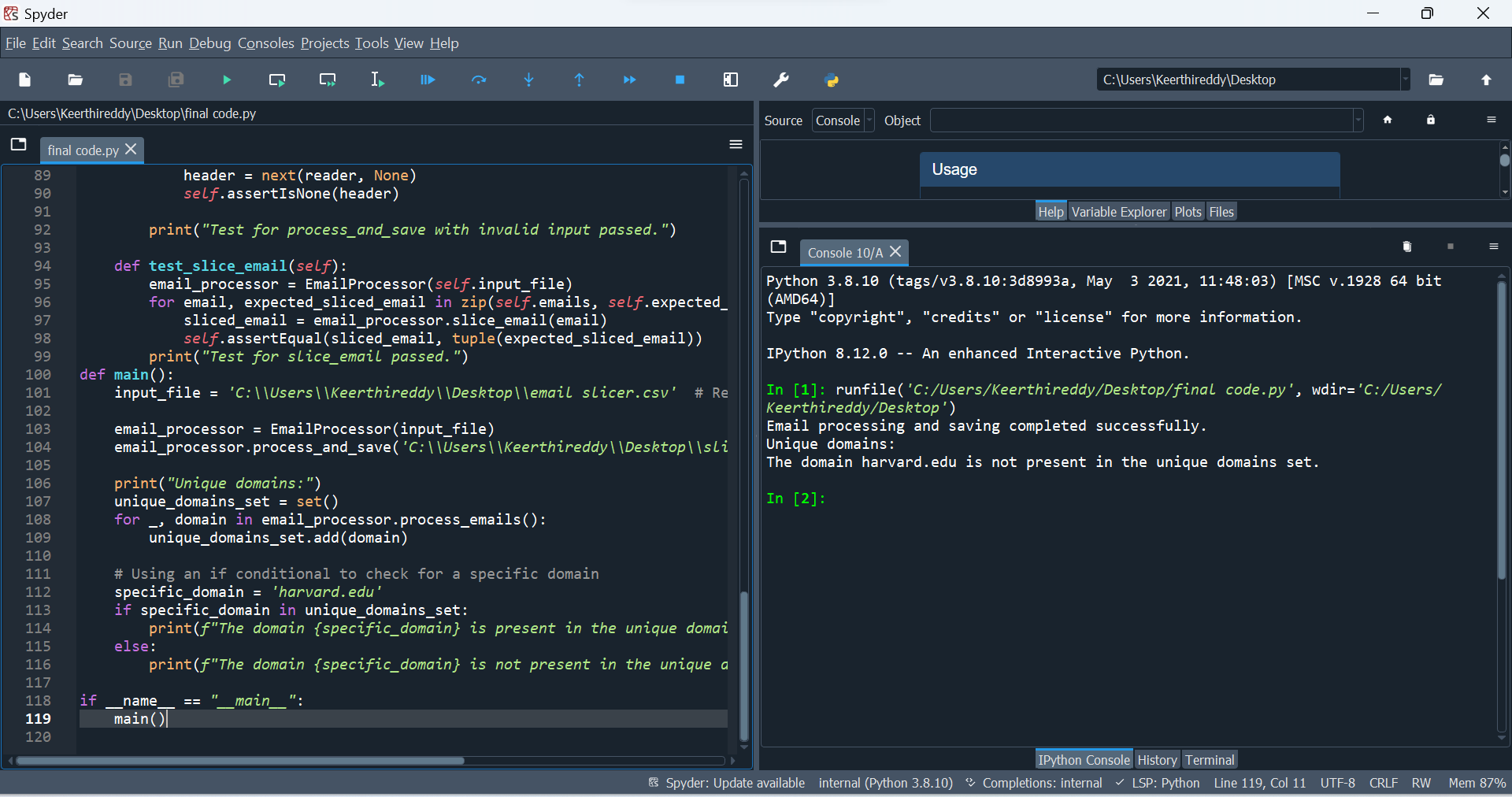
After collecting the unique domains in the set, an if conditional is used to check if a specific domain (**'harvard.edu'**) is present in the unique domains set.

If the domain is present, it prints a message indicating its presence.

If the domain is not present, it prints a message indicating its absence.

**if \_\_name\_\_ == "\_\_main\_\_":**: This is a common Python idiom that checks if the script is being run as the main program (not imported as a module). If it is the main program, the **main()** function is called to execute the program.

*After execution~*



***Conclusion:***

The Email Slicer project in Python is a simple program that takes an email ID as input and performs slicing operations to extract and output both the username and the domain of the email ID. The project is highly straightforward, quick to complete, and requires no additional setup.

Additional features can be incorporated into the application, including a validation check for valid email addresses. Regular expressions can be employed to validate the format of email addresses for this purpose.

Furthermore, a functionality allowing users to input multiple email addresses simultaneously for batch slicing can also be integrated.

This project holds applicability across various sectors and disciplines, such as marketing and data research. For instance, in marketing, sales, and client segmentation efforts, it can be utilized to group and sort email addresses based on their domain names.

Numerous organizations can adopt this initiative to enhance productivity and efficiency. For example, it can be employed to automatically extract email addresses from extensive datasets, serving diverse purposes.

~Thank You~