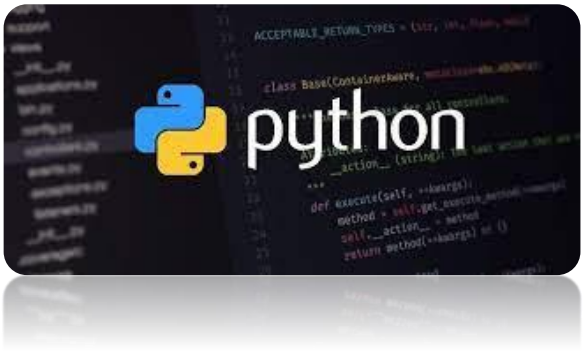
Welcome to the new mini project

" Securely Streamline OTP Retrieval "



Author:

Vivek Jangam

Step 1 :

```
# import dependencies
import imaplib                    #for accesing email via imap server
import email                      #for parsing email messages
from email.header import decode_header  #access email headers
import re                         #to create otp pattern for search
```

[links 1 : Know more about Imaplib]

[Link 2 : Know more about email module]

[link 3 : Know more about import re module]

Step 2 :

```
# now fetch the login details from CSV file one by one
with open("data.txt", "r") as file:
    for i in file:
        data = i.split(",")
        imap_server = data[0]
        username = data[1]
```

password = data[2]

Link4 : know more about with open in file Handeling

Note:

- keep your details in CSV file format for eg. -> Imap_server,username,password
- make name of your csv file as 'data.txt'
- Update imap_server with your imap_server
- Update username with your mail_id
- Update password as your mail_id password

Step 3:

#connect to imap server

mail = imaplib.IMAP4_SSL(imap_server)

Step 4:

#login to email account

mail.login(username, password)

Step 5 :

Select the inbox

mail.select("inbox")

Link 5 : know how mail.select("inbox") works

Step 6:

#search for emails

result, data = mail.search(None, "ALL")

Link 6: Know How mail.search() works and what values it returns

Link 7: Know more about what search returns

Note : print result and data to see what actually it returns

- print(result)
- print(data)

Step 7 :

segregate email ids to iterate through it one by one

email_ids = data[0].split()

email_ids

then create empty list to store otps

otps = []

Step 8:

Iterate through the list of email IDs

for mail_id in email_ids:

    #fetch the email using it's id

    result, raw_email = mail.fetch(mail_id, "(RFC822)")

    Link8: Know how mail.fetch(mail_id, "(RFC822)") works and what it returns

    Link9: Know about RFC822 format

    #parse the email data

    email_message = email.message_from_bytes(raw_email[0][1])

    Link 10: Know How Above method works and what it returns

```
 # Extract email details
subject = email_message["Subject"]
sender = email.utils.parseaddr(email_message["From"])[1]
```

Link 11: Know How utils.parseaddr(email_message["From"])[1] works

```
#Print email details
print("Subject:", subject)
print("Sender:", sender)

body = ""

# Get email body if exists
 if email_message.is_multipart():
        for part in email_message.walk():
        if part.get_content_type() == "text/plain":
         body = part.get_payload(decode=True).decode()
         break
 else:
                body = email_message.get_payload(decode=True).decode()

  # Check if OTP exists in the email body
  otp_match = re.search(r'\b\d{6}\b|\b\d{3}\s\d{3}\b', body)
 # in above code we created a pattern which will search for exact 6 digits or exact 6 digits with space      after first three digits
  if otp_match:
```

```
        otp = otp_match.group()
        #and then append it in a empty list “otps”  which we created in step 7
        otps.append(otp)
```

Note :

- Print list otps to see which values it stored
- Print(otps)

Step 9 :

```
# now we will print only latest otp by using reverse indexing of list like below
if otps:
    print("Found OTP:", otps[-1])
else:
    print("No OTPs found")
```

Step 10 :

```
#close the connection
mail.logout()
```

Yay ! You Just Learned How to access Mails and fetch only otps out of it

Documentation

Q.1) What is Import Imaplib ? `import imaplib` is a Python statement used to import the `imaplib` module.

Functionality:

- The `imaplib` module is part of Python's standard library and provides functionality for working with Internet Message Access Protocol (IMAP) mailboxes.

- IMAP is a protocol used by email clients to retrieve emails from a mail server. The `imaplib` module allows Python programs to interact with IMAP servers to perform various email-related tasks, such as fetching, searching, and manipulating email messages.

Common Use Cases:

1. **Fetching Email Messages**: With `imaplib`, you can connect to an IMAP server and fetch email messages from a mailbox.

2. **Searching for Emails**: It allows you to search for specific email messages based on criteria such as sender, recipient, subject, or date.

3. **Managing Mailboxes**: You can create, rename, or delete mailboxes on the server using `imaplib`.

4. **Handling Email Flags**: `imaplib` provides functions to set, unset, or query flags associated with email messages, such as "seen" or "deleted".

5. **Handling MIME Messages**: You can work with MIME (Multipurpose Internet Mail Extensions) messages, including parsing and decoding attachments.

Example Usage:

```python

import imaplib


# Connect to the IMAP server

mail = imaplib.IMAP4_SSL('imap.example.com')


# Log in to the server

mail.login('username', 'password')


# Select a mailbox
```

```
mail.select('INBOX')

# Fetch email messages
result, data = mail.search(None, 'ALL')

# Process fetched email messages
for num in data[0].split():
    result, message_data = mail.fetch(num, '(RFC822)')
    # Process message_data...

# Log out from the server
mail.logout()
```

In this example, `imaplib` is used to connect to an IMAP server, log in, select a mailbox, fetch email messages, process them, and then log out from the server. This is a common workflow for accessing email messages using Python.

(Go to start)

Q.2) How import email module works ?

`import email` is a Python statement used to import the `email` module.

Functionality:

- The `email` module is part of Python's standard library and provides functionality for working with email messages, including parsing, composing, and manipulating emails in various formats.

- It supports parsing and generating messages in both MIME (Multipurpose Internet Mail Extensions) format and RFC 2822 format (an updated version of RFC 822).

Common Use Cases:

1. **Parsing Email Messages**: With the `email` module, you can parse email messages from strings or file objects, extracting various components such as headers, body, and attachments.

2. **Composing Email Messages**: You can create new email messages, set headers, add attachments, and compose multipart messages using the classes provided by the `email` module.

3. **Manipulating Message Headers**: It allows you to access and modify headers of email messages, such as sender, recipient, subject, and date.

4. **Working with MIME Messages**: The `email` module provides support for working with MIME messages, including multipart messages and various content types such as text/plain, text/html, and attachments.

(Go to start)

Q.3) what is ' import re '/ How ' regular expression ' works ?

`import re` is a Python statement used to import the `re` module.

Functionality:

- The `re` module is part of Python's standard library and provides support for working with regular expressions (regex).

- Regular expressions are powerful tools for matching patterns in strings, allowing for complex text manipulation and searching.

Common Use Cases:

1. **Pattern Matching**: The `re` module allows you to define patterns and search for matches within strings.

2. **String Manipulation**: You can use regular expressions to perform string manipulation tasks such as substitution, splitting, and extraction.

3. **Validation**: Regular expressions are often used for validating input, such as checking whether a string conforms to a specific format (e.g., email address, phone number).

4. **Text Parsing**: They are useful for parsing and extracting information from structured or semi-structured text data.

5. **Data Cleaning**: Regular expressions can help clean and preprocess text data by removing unwanted characters, formatting inconsistencies, or noise.

Example Usage:

```python
import re

# Define a regular expression pattern
pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

# Search for email addresses in a string
text = "Contact us at info@example.com or support@example.org"
matches = re.findall(pattern, text)

# Print the matches
for match in matches:
    print("Email:", match)
```

In this example, `re.findall()` is used to search for email addresses in a given string `text` based on the defined regular expression pattern. The matches are then printed to the console.

(Go to start)

Q.4) How ' with open ' works in file Handeling ?

The `with open()` statement in Python is used to open files in a way that ensures they are properly closed after being used, even if an exception occurs during the execution of the code block. This is commonly referred to as a context manager.

Functionality:

- `open()` is a built-in function in Python used to open files for reading, writing, or appending data.

- When used with the `with` statement, `open()` returns a file object for the specified file, and this file object is assigned to a variable. The file is automatically closed at the end of the `with` block.

Syntax:

```python
with open(file_path, mode) as file_object:
    # Code block to work with the file_object
```

- `file_path`: Path to the file to be opened.

- `mode`: Optional parameter specifying the mode in which the file should be opened (e.g., `'r'` for reading, `'w'` for writing, `'a'` for appending). If not provided, the default mode is `'r'` (reading).

Use Cases:

1. **Reading from Files**:

```python
with open('example.txt', 'r') as file:
    data = file.read()
    print(data)
```

2. **Writing to Files**:

```python
with open('example.txt', 'w') as file:
    file.write('Hello, world!')
```

3. **Appending to Files**:

```python
with open('example.txt', 'a') as file:
    file.write('\nNew line added!')
```

Benefits:

- **Automatic File Closure**: The file is automatically closed at the end of the `with` block, even if an exception occurs during the execution of the code inside the block.

- **Simplified Syntax**: It provides a more concise and readable syntax compared to manually opening and closing files using `try`...`finally` blocks.

Example:

```python
```

```
# Reading from a file using with open

with open('example.txt', 'r') as file:

    data = file.read()

    print(data)
```

In this example, the file named `example.txt` is opened for reading (`'r'` mode) using the `with open` statement. Inside the `with` block, the content of the file is read using `file.read()`, and then printed to the console. After the execution of the block, the file is automatically closed.

(Go to start)

Q.5) How ‘ mail.select("inbox") ‘ works and what it returns ?

The line `mail.select("INBOX")` in the code connects to the IMAP server and selects the mailbox named "INBOX". Here's what it does and what output it returns:

1. **Selecting the Mailbox ("INBOX"):**

- When you connect to an IMAP server, you're typically connecting to a specific mailbox on that server. The "INBOX" is a special mailbox that holds incoming emails.

- By selecting the "INBOX" mailbox, you're indicating that you want to perform operations (such as fetching, searching, or storing emails) on the emails in this mailbox.

2. **Return Value:**

- The `select()` method returns a tuple containing two elements: a response code and a data string.

- The response code is a string indicating the result of the operation. It can be one of the following:

  - "OK": The operation was successful.

- "NO": The operation failed.

- "BAD": The operation is not recognized or is invalid.

- The data string contains additional information about the selected mailbox, such as the number of messages in the mailbox, the mailbox's UID validity, etc.

Here's an example of how you might use the `select()` method and handle its return value:

```python
result, data = mail.select("INBOX")

if result == 'OK':
    print("Mailbox selected successfully.")
    print("Number of messages in INBOX:", data[0])
else:
    print("Failed to select mailbox: ", data)
```

In this example, if the result of the `select()` operation is "OK", it prints a success message and the number of messages in the "INBOX" mailbox. Otherwise, it prints a failure message along with the reason for the failure.

(Go to start)

Q.6) How ‘ result, data = mail.search(None, "ALL") ‘ works and what It returns ?

The line `result, data = mail.search(None, "ALL")` in the code you provided searches for email messages in the currently selected mailbox. Here's what it does and what value it returns:

1. **Searching for Email Messages:**

- The `search()` method of the `imaplib` library is used to search for email messages in the currently selected mailbox.

- In the line `mail.search(None, "ALL")`, the first argument (`None`) specifies the character set to use for searching (it's typically `None` for the default character set).

- The second argument (`"ALL"`) specifies the search criteria. In this case, `"ALL"` means you want to search for all email messages in the selected mailbox.

2. **Return Value:**

- The `search()` method returns a tuple containing two elements: a response code and a data string.

- The response code is a string indicating the result of the search operation. It can be one of the following:

  - "OK": The search operation was successful.

  - "NO": The search operation failed.

  - "BAD": The search operation is not recognized or is invalid.

- The data string contains the search results, which are the IDs of the email messages that match the search criteria. These IDs are space-separated and represented as a single string.

Here's an example of how you might use the `search()` method and handle its return value:

```python
result, data = mail.search(None, "ALL")

if result == 'OK':
    print("Search operation was successful.")
    print("Email IDs of matching messages:", data[0])
```

```
else:
    print("Search operation failed:", data)
```

In this example, if the result of the `search()` operation is "OK", it prints a success message along with the email IDs of the matching messages. Otherwise, it prints a failure message along with the reason for the failure.

(Go to start)

Q.7) Explain more about argument in search method and other use case

The `search()` method in the `imaplib` library is used to search for email messages in the currently selected mailbox on an IMAP server. It takes search criteria as arguments to filter the messages according to specific conditions. Let's delve into its arguments and explore some common use cases:

Arguments of `search()` method:

1. **Charset (optional)**:

- The first argument specifies the character set to use for searching. It's typically `None`, indicating the default character set. You can specify a particular character set if needed.

2. **Search Criteria**:

- The second argument specifies the search criteria to filter the email messages. It can be one or more criteria combined using logical operators (e.g., "AND", "OR", "NOT").

Common Search Criteria:

1. **ALL**:

- `"ALL"` retrieves all email messages in the mailbox.

2. **UNSEEN**:

- `"UNSEEN"` retrieves email messages that have not been read (marked as unseen).

3. **SEEN**:

- `"SEEN"` retrieves email messages that have been read (marked as seen).

4. **FROM**:

- `"FROM sender@example.com"` retrieves email messages sent from a specific sender.

5. **TO**:

- `"TO recipient@example.com"` retrieves email messages sent to a specific recipient.

6. **SUBJECT**:

- `"SUBJECT keyword"` retrieves email messages with a specific subject.

7. **BEFORE**:

- `"BEFORE date"` retrieves email messages received before a specific date.

8. **SINCE**:

- `"SINCE date"` retrieves email messages received since a specific date.

Use Cases:

1. **Fetch Unread Messages**:

   - Use `mail.search(None, "UNSEEN")` to fetch unread email messages from the mailbox.

2. **Search by Sender**:

   - Use `mail.search(None, "FROM sender@example.com")` to search for email messages from a specific sender.

3. **Search by Subject**:

   - Use `mail.search(None, "SUBJECT keyword")` to search for email messages with a specific subject.

4. **Search by Date Range**:

   - Use `mail.search(None, "SINCE 01-Jan-2023 BEFORE 31-Jan-2023")` to search for email messages received within a specific date range.

Example:

```python
# Connect to the IMAP server and select the mailbox
mail = imaplib.IMAP4_SSL('imap.example.com')
mail.login('username', 'password')
mail.select("INBOX")

# Search for email messages sent by a specific sender
result, data = mail.search(None, "FROM sender@example.com")

if result == 'OK':
```

```
    # Process the email IDs of matching messages
    for email_id in data[0].split():
        # Fetch and process each email message
        # (code to fetch and process the email message goes here)
else:
    print("Search operation failed:", data)
```

In this example, we connect to the IMAP server, select the "INBOX" mailbox, and search for email messages sent by a specific sender (`sender@example.com`). We then process the email IDs of the matching messages, which can be used to fetch and process each email message individually.

(Go to start)

Q.8) ‘ result, raw_email = mail.fetch(mail_id, "(RFC822)") ‘ what this method does and what it returns ?

The line `result, raw_email = mail.fetch(mail_id, "(RFC822)")` in the code is used to fetch the raw content of an email message from the IMAP server. Here's what it does and what it returns:

Functionality:

- The `fetch()` method of the `imaplib` library is used to retrieve email messages or parts of email messages from the currently selected mailbox on the IMAP server.

- It takes the unique identifier (UID) of the email message to fetch and a set of message parts to fetch as arguments.

- The message parts to fetch are specified using a format string. In this case, `"(RFC822)"` specifies that the entire email message should be fetched in its raw form.

Return Value:

- The `fetch()` method returns a tuple containing two elements: a response code and a list of message data items.

- The response code indicates the result of the fetch operation, such as whether it was successful or not.

- The list of message data items contains the fetched data for each specified message part.

- In this case, `raw_email` will contain the raw content of the fetched email message, including its headers and body, in RFC 822 format.

Example Usage:

```python
result, raw_email = mail.fetch(mail_id, "(RFC822)")

if result == 'OK':
    # Process the fetched email message
    print("Raw Email Content:")
    print(raw_email[0][1])  # Accessing the raw email content
else:
    print("Failed to fetch email:", raw_email)
```

In this example, if the result of the `fetch()` operation is "OK", it prints the raw content of the fetched email message. Otherwise, it prints a failure message along with the reason for the failure. The raw content of the email message is accessed using `raw_email[0][1]`, where `raw_email[0]` is a tuple containing the fetched data, and `raw_email[0][1]` accesses the second element of that tuple, which contains the raw email content.

(Go to start)

Q.9) Explain more about rfc822 format

RFC 822 is a specification that defines the format of Internet text messages. It was published by the Internet Engineering Task Force (IETF) in August 1982 and is now obsoleted by RFC 5322, which is an update that clarifies certain aspects of the original specification.

However, the term "RFC 822" is often used colloquially to refer to the format of email messages in general, including those conforming to both RFC 822 and RFC 5322.

Here are some key points about RFC 822:

1. **Message Structure**: RFC 822 defines the structure of email messages, including headers and body. Headers contain metadata about the message, such as the sender, recipient, subject, date, and other optional fields. The body contains the main content of the message.

2. **Header Fields**: RFC 822 specifies a set of header fields that can appear in email messages. Common header fields include "From", "To", "Subject", "Date", "Message-ID", "Content-Type", and "Content-Disposition", among others.

3. **Syntax Rules**: RFC 822 defines the syntax rules for constructing email addresses, dates, and other elements within the message. For example, it specifies how to format email addresses ("local-part@domain"), how to represent dates and times in the "Date" header field, and how to encode and represent message bodies.

4. **Message Parsing**: When working with email messages in programming, the term "RFC 822 format" often refers to the standard way of representing email messages as text, following the conventions specified in RFC 822. This format is used for parsing and generating email messages in various email-related protocols and libraries.

In the context of the `imaplib` library in Python, the `"RFC822"` argument passed to the `fetch()` method indicates that the entire email message should be fetched in its raw form,

following the format specified by RFC 822. This includes all header fields, such as "From", "To", "Subject", "Date", etc., as well as the message body.

(Go to start)

Q.10) ‘ email_message = email.message_from_bytes(raw_email[0][1]) ‘ what this do and what it returns ?

In the line `email_message = email.message_from_bytes(raw_email[0][1])`, the `email.message_from_bytes()` function is used to parse the raw email data fetched from the IMAP server. Here's what it does and what it returns:

Functionality:

- The `email.message_from_bytes()` function is part of the `email` module in Python, which provides tools for parsing and composing email messages.

- It takes a bytes-like object containing the raw email data as input.

- This function parses the raw email data and constructs a `Message` object representing the email message.

Return Value:

- The `Message` object represents the email message and provides access to its various components, such as headers and body.

- It's a hierarchical data structure where different parts of the email message (e.g., headers, body, attachments) are accessible through attributes and methods of the `Message` object.

Example Usage:

```python

# Parse the email data

email_message = email.message_from_bytes(raw_email[0][1])

# Access email components
```

```
print("From:", email_message["From"])

print("To:", email_message["To"])

print("Subject:", email_message["Subject"])

print("Date:", email_message["Date"])

print("Message Body:")

print(email_message.get_payload())
```

In this example, after parsing the raw email data using `email.message_from_bytes()`, you can access various components of the email message (such as sender, recipient, subject, date, and body) using attributes and methods of the `Message` object (`email_message`). The `get_payload()` method retrieves the payload of the email message, which can include the main text content or attachments, depending on the message structure.

(Go to start)

Q.11) Email.utils.parseaddr(email_message["From"])[1] - What it does and what it returns ?

email.utils.parseaddr(email_message["From"])

This retrieves the "From" header field from the email message. which contain sender's Name & address

email.utils.parseaddr(), this will convert both values into tuple like (display_name, email_address)

so use index 1 to fetch email_adress and index 0 to fetch sender's Name

Details :

In the line `sender = email.utils.parseaddr(email_message["From"])[1]`, the `email.utils.parseaddr()` function is used to parse the "From" header field of an email message. Here's what it does and what it returns:

Functionality:

- The `email.utils.parseaddr()` function is part of the `email.utils` module in Python's standard library. It's used to parse an email address or header field value into a tuple containing the display name and the email address parts.

- It takes a string representing an email address or an email header field value as input.

- This function parses the input string and extracts the display name and email address parts, if present.

Return Value:

- The return value of `email.utils.parseaddr()` is a tuple containing two elements: the display name and the email address.

- In the line `sender = email.utils.parseaddr(email_message["From"])[1]`, the `[1]` index is used to access the second element of the tuple, which represents the email address part.

- Therefore, `sender` will contain the email address extracted from the "From" header field of the email message.

Example Usage:

```python

# Parse the email address from the "From" header field

sender = email.utils.parseaddr(email_message["From"])[1]

```

In this example, after parsing the "From" header field of the email message using `email.utils.parseaddr()`, the email address part (the second element of the tuple) is extracted and assigned to the variable `sender`. This allows you to obtain the email address of the sender of the email message.

(Go to start)

Your Module To access emails end fetch otp out of it ends here !!!

Regards,
Vivek Jangam
Skilled Python developer
Contact 9834747047
Mail vivekjangam005@gmail.com

Thank You